Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática

INTERFACE DÍODO

Hugo David Martins Sousa

DISSERTAÇÃO

MESTRADO EM SEGURANÇA INFORMÁTICA

2013

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática

INTERFACE DÍODO

Hugo David Martins Sousa

DISSERTAÇÃO

MESTRADO EM SEGURANÇA INFORMÁTICA

Dissertação orientada pelo Prof. Doutor Paulo Jorge Esteves Veríssimo

2013

Agradecimentos

Durante o percurso que conduziu a esta dissertação, incluindo todos os anos como estudante da Faculdade de Ciências da Universidade de Lisboa, foram surgindo pessoas que assumiram um papel preponderante na minha vida, tanto profissional como pessoal.

Começo por agradecer ao Professor Doutor Paulo Jorge Esteves Veríssimo, professor e orientador, o acompanhamento, clareza e ajuda no presente trabalho. O seu sucesso no meio académico pode e deve ser um ponto de motivação para todos os estudantes.

Em segundo lugar menciono com orgulho e um enorme prazer de ter tido a oportunidade de conhecer alguns investigadores e professores da Faculdade de Ciências. Obrigado ao Diego Kreutz pela ajuda e suporte na realização deste trabalho. Prof. Jorge Buescu, a sua alegria, entusiasmo e esforço incansável por cativar os alunos pela Matemática foi um ponto de viragem no meu desempenho académico. É a prova viva que não precisamos de aprofundar relações para nos sentirmos inspirados. Ao Prof. Alysson Bessani, cujas aulas são sempre uma fonte de conhecimento e por estar sempre disponível para resolução de problemas. Ao Prof. António Casimiro, a sua forma de relacionamento e interação mantém-nos conscientes que a Faculdade é mais do que um local de estudo e trabalho. Agradeço à Prof. Dulce Domingos, coordenadora do Mestrado em Segurança Informática pelo esforço de tornar este Mestrado numa área de referência. O presente é importante, mas mais importante é o futuro.

Agradeço a todos os meus companheiros e colegas de Licenciatura e Mestrado. Uma palavra de apreço e um profundo obrigado. Acredito que se fomos bons, muito se deveu à nossa entreajuda e convivência. Um especial obrigado ao Rúben Campos, Emanuel Alves, Cristiano Iria, André Matias e Rúben Costa. Ao Lab 25, local de trabalho dos investigadores juniores do grupo Navigators onde estive inserido, um obrigado do fundo do coração e continuo a achar que todos juntos nenhuma outra empresa teria hipótese.

À minha afilhada Leonor, que embora estes cinco anos profissionais me tenham retirado tempo de convivência, a alegria e o sentimento que me inunda quando estamos juntos é um motor de motivação para fazer mais e melhor. Sei que um dia também farás tudo para ser a melhor e eu estarei cá para te ajudar.

Longe de serem os últimos, o maior obrigado à minha família. À minha prima Carla Almeida pelo exemplo, primo Bruno Almeida e Ricardo Almeida pelos momentos alegres nos períodos de descontração. Obrigado aos meus tios e primos que estão longe do nosso Portugal, a vossa distância e sucesso permite-me alargar horizontes e nunca perder a esperança.

Por fim, não podia estar mais alegre por viver no expoente máximo da evolução tecnológica. É fantástico ver coisas novas e criadas por pessoas não menos espetaculares a surgir todos os dias.

Aos meus pais, irmão e Inês.

Resumo

Com o constante crescimento dos serviços *online*, as interações realizadas por utilizadores ou servidores que se encontram numa rede com baixo nível de segurança (e.g. Internet) com utilizadores ou serviços que se encontram num domínio de alto nível de segurança (e.g. servidor de e-mail) têm vindo a aumentar. Além desta interação entre redes com diferentes níveis de segurança, dentro do *datacenter* da mesma organização existem interações entre entidades que operam com níveis de segurança distintos, sem que se considerem verdadeiramente inseguros. Por outro lado, devido aos protocolos existentes, muitas das interações são caraterizadas por trocas de dados em ambos os sentidos da ligação (e.g. *Transmission Control Protocol*). Não obstante, existem serviços cuja lógica aplicacional (e-mail, sistemas de votação *online*, etc.), embora assentem sobre estes protocolos bidirecionais, não justifica a necessidade de tráfego nos dois sentidos em simultâneo, tratando-se, muitas vezes, de informações confidenciais restritas a certos utilizadores.

Nesta dissertação apresentamos a Interface Díodo, um dispositivo de rede tolerante a faltas Bizantinas, que restringe a passagem de tráfego a um sentido da ligação, bloqueando a informação no sentido oposto. O serviço fornecido pode ser utilizado por vários clientes em simultâneo para diferentes serviços finais. Definimos uma arquitetura, um protótipo e os respetivos resultados obtidos, independentemente do protocolo de transporte (unidirecional ou bidirecional). Com a Interface Díodo é possível a troca de dados entre uma rede com um nível de segurança inferior para um nível de segurança superior, ou vice-versa, mas nunca nos dois sentidos. Se o fluxo de tráfego acontecer de uma rede insegura para uma rede segura, garante-se a confidencialidade dos dados no domínio superior segurança. No sentido oposto, garante-se a integridade dos dados no domínio superior de segurança.

Palavras-chave: comunicação unidirecional, comunicação hierárquica, díodo de dados, rede unidirecional, tolerância a faltas

Abstract

Online services provided in the business world are increasing the probability of interactions between servers and users on insecure network (e.g. Internet) and services and users in high security domains (e.g. e-mail server). Besides these interactions between insecure and secure networks, there are interactions that happen inside datacenters made by entities operating in different kind of security domains, without any of them being completely inscure. On the other hand, due to the way current protocols operate, there are interactions characterized by data exchanged in both directions (e.g. Transmission Control Protocol). Some services, such as e-mail or online voting systems, while using these protocols, do not justify the need for traffic in both directions, specially in sensitive information that should remain in one place and only accessed by specific users.

In this report we introduce the Diode Interface, a network device that allows the traffic in a connection to flow just in one direction, blocking any sort of information coming from the opposite side. The service provided can be used by several clients sending data to different final services simultaneously. We define an architecture, prototype and obtained results, regardless of transport protocol (unidirectional or bidirecional) which transfer data between a network and the diode. With the Diode Interface it is possible the data exchange between a low security network and high security network or vice versa, but never in both directions at the same time. Allowing traffic only to flows from a low security domain to a high security domain, one guarantees the confidentiality of critical data. Allowing the traffic to flow from the high security level to low security level one guarantees data integrity that is in the high security domain.

Keywords: unidirectional communication, hierarchical communication, data diode, unidirectional network, fault tolerance

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Esta é a era da Internet das Coisas [1], e com a vasta transição de serviços para o mundo *online*, os desafios associados [2, 1] aos mecanismos de segurança também aumentam. A atual dependência destes sistemas obriga ao seu correto funcionamento, deixando de ser apenas necessário para ser obrigatório.

A abordagem tomada sobre a construção de sistemas seguros foi variando ao longo do tempo. Até há relativamente pouco tempo, as grandes preocupações de segurança eram solucionadas através da utilização de dispositivos de rede que diminuíssem a probabilidade de ocorrer um ataque bem-sucedido. As *firewalls*, que impedem ou permitem a passagem de tráfego para determinada entidade, as *proxies*, com um papel idêntico mas fazendo uma avaliação ao tipo de conteúdo que se pretende transmitir. As cifras têm como objetivo manter a confidencialidade da informação em trânsito, mas uma vez comprometida a chave de cifra ou é realizado um ataque bem sucedido ao algoritmo de cifra, perde-se a confidencialidade (um dos algoritmos de cifra mais seguros, o RSA, foi quebrado há relativamente pouco tempo). O problema desta abordagem é que uma falha em qualquer um destes componentes faz com que todo o sistema fique à mercê do atacante. Essencialmente, projetavam-se sistemas que definissem o vetor de ataque [1] de forma clara e concisa. Com o passar do tempo, que nos traz até aos dias de hoje, a abordagem mudou. Hoje, mais que evitar ataques, é obrigatório sobreviver a estes, transformando totalmente a perspetiva de sistema seguro. Isto significa que se assume à partida a possibilidade de existi-

[1] Os meios utilizados por um utilizador malicioso para atacar e ganhar acesso a um computador ou rede com o objetivo de realizar ações maliciosas.

rem ataques bem-sucedidos e os esforços devem assentar sobre a construção de sistemas que sobrevivam a estes. Daí nasce o conceito de Tolerância a Faltas [3] e Tolerância a Faltas Bizantinas [4].

Além da necessidade de tolerar e sobreviver a potenciais comportamentos maliciosos, e devido à complexidade crescente dos sistema de informação, existem comunicações entre domínios de baixo nível de segurança (BNS), por exemplo a Internet, e domínios de alto nível de segurança (ANS), como um repositório de *logs*. Dentro da mesma organização, inclusive, existem interações entre redes com diferentes domínios de segurança. Por essas razões, é conveniente que alguns componentes constituintes de uma rede permaneçam isolados dos demais. Porém, os componentes que se encontram num alto nível de segurança são utilizados por serviços, utilizadores e pessoal administrativo que se encontram em domínios com níveis de segurança inferiores. Quando entidades num domínio de segurança inferior geram *logs*, estes têm interesse para auditorias e deteção automática de problemas em determinados constituintes da rede. É informação valiosa e, como tal, deve ser mantida numa rede de alta segurança, evitado, por exemplo, que um utilizador malicioso realize um ataque com sucesso e apague o seu rasto.

Abordando o fluxo de dados e a essência dos protocolos de transporte utilizados na Internet, que convém salientar a sua importância para o tema desta dissertação, existem diversos serviços de Internet que são construídos para uma comunicação bidirecional (como um pedido por parte do cliente e uma resposta do servidor), não sendo menos verdade que a lógica aplicacional de alguns serviços é de cariz unidirecional. Um desses exemplos é um sistema de votação eletrónica. Um eleitor realiza o seu voto que é posteriormente armazenado num servidor. Não existe qualquer informação no sentido oposto. Um sistema de *logs* é outro exemplo de uma aplicação unidirecional. Uma máquina gera um *log* que é enviado e guardado num repositório específico para o efeito.

É neste cenário que se contextualiza o conceito de díodo. O díodo é um componente eletrónico que quando inserido num circuito eletrónico impede que a corrente elétrica circule nos dois sentidos. Estes díodos são polarizados de forma direta ou inversa. Na polarização direta, o díodo está colocado no sentido convencional da ligação; na polarização inversa, o díodo está no sentido "negativo"para "positivo"da ligação. Transpondo o díodo da eletrónica para o mundo tecnológico (díodo de dados), significa que existe um componente de rede que

quando polarizado de forma direta permite a passagem de tráfego nesse sentido, e quando inversamente polarizado, impede a passagem de tráfego. Desta forma, consegue-se concretizar um protocolo que estabeleça uma ligação unidirecional, uma vez que só existe tráfego num sentido.

Esta dissertação foi concebida no âmbito do Projeto em Engenharia Informática (PEI) do Mestrado em Segurança Informática (MSI) da Faculdade de Ciências da Universidade de Lisboa.

O trabalho apresenta a Interface Díodo, um díodo de dados concretizado em *software*, que permite a passagem de dados apenas num sentido da ligação. O serviço fornecido pode ser utilizado por vários clientes em simultâneo para diferentes serviços finais. A Interface Díodo é tolerante a faltas Bizantinas e aceita como entrada dados transportados tanto por protocolos unidirecionais (UDP) como bidirecionais (TCP). A concretização da Interface Díodo utiliza maioritariamente tecnologias já desenvolvidas e bem conhecidas combinadas numa arquitetura inovadora. Nesta secção será abordada a motivação por trás do trabalho desenvolvido, os objetivos pretendidos na sua elaboração e a estrutura deste documento.

1.1 Motivação

O trabalho apresentado deve a sua motivação às seguintes constatações:

- Embora alguns protocolos de comunicação, como o caso do TCP, operem numa lógica de comunicação bidirecional, algumas aplicações têm uma lógica de cariz unidirecional:
  - **Votação online:** o eleitor submete o seu voto;
  - **Transferência de e-mail:** é enviado um e-mail e tal não implica uma resposta;
  - **Sistema de *logs*:** são gerados registos de *log* em determinada máquina que são guardados num servidor de *logs*;
  - **Acessos para confirmação de dados a base de dados:** a confirmação de *username* e *password* não necessita de uma resposta que contenha dados, apenas se a combinação é verdadeira ou falsa;

  - **Transferência de ficheiros:** o envio de ficheiros para um servidor de ficheiros;

- Necessidade de aumentar a segurança na comunicação entre redes inseguras e redes seguras;

- A atual solução para um díodo de dados não é tolerante a faltas e tem como desvantagem os problemas associados à utilização de fibra ótica;

- Desenvolvimento de *plugins* para interação entre dois domínios de segurança através da Interface Díodo. Estes *plugins* concretizam a nível aplicacional toda a lógica de díodo, criando-se desta forma o protocolo Interface Díodo. Por exemplo, um *plugin* para *browser* onde o utilizador define um pedaço de informação a ser enviado para uma rede de alta segurança.

1.2 Contribuições

Esta dissertação apresenta a Interface Díodo, um dispositivo de rede que medeia a comunicação entre duas entidades que se encontram em domínios com diferentes níveis de segurança.

Sinteticamente, este trabalho visa contribuir com uma Interface Díodo que garante:

(a) **Modelo *publish/subscribe*:** o modo de operação da interface segue o comportamento típico do modelo *publish/subscribe*;

(b) **Independência do protocolo de transporte:** a interface é independente do protocolo de transporte. Correto funcionamento tanto com TCP como com UDP;

(c) **Comunicação unidirecional:** existe um cliente (*publisher*) que produz informação que é escrita num servidor final (*subscriber*). Um produtor não pode ser simultaneamente um consumidor e vice-versa;

(d) **Tolerância a ataques:** a interface pode ser atacada e deve continuar a operar corretamente;

(e) **Confidencialidade e Integridade**: os dados armazenados na entidade de alto nível de segurança são confidenciais, no caso do díodo diretamente polarizado, e íntegros no caso do díodo inversamente polarizado.

1.3 Estrutura do Documento

Este documento encontra-se estruturado da seguinte forma:

Capítulo 2 Descrição do trabalho relacionado e introdução aos conceitos de díodo e aos seus constituintes;

Capítulo 3 Apresentação do modelo do sistema, arquitetura e respetiva descrição, bem como as tecnologias utilizadas;

Capítulo 4 Definição e explicação dos detalhes de concretização, tais como configurações de rede, objetos próprio da aplicação e dos componentes que compõem a Interface Díodo.;

Capítulo 5 Descrição dos testes realizados, discussão dos resultados obtidos e desafios e limitações da solução;

Capítulo 6 Introdução do trabalho a realizar futuramente e conclusão finais.

Capítulo 2

Estado da Arte

Este capítulo introduz os principais conceitos, como tolerância a faltas Bizantinas, protocolos unidirecionais e bidirecionais, bem como alguns dos principais trabalhos realizados relativos ao díodo, como é o caso do díodo de dados [5] e o *Pump* [6, 7].

2.1 Protocolos Unidirecionais e Bidirecionais

A Internet é construída sobre um modelo concetual denominado por *Open System Interconnection* (OSI) [8]. Este modelo, representado na figura 2.1, define um *standard* para as comunicações realizadas na rede global, e é composto por sete camadas: física, ligação de dados, rede, transporte, sessão, apresentação e aplicação. A função de cada uma destas camadas é a seguinte:

- **Camada física** - responsável pelo envio e receção de sinais elétricos, representativos dos bits de informação. Por exemplo, Ethernet;

- **Camada de ligação dados** - responsável pelo endereçamento físico de cada bit que é transmitido. Por exemplo, *Logical Link Control* (LLC) e *Media Access Control* (MAC);

- **Camada de rede** - responsável por determinar qual o caminho lógico pelo qual os bits têm que passar até chegar ao destino. Por exemplo, *Internet Protocol* (IP);

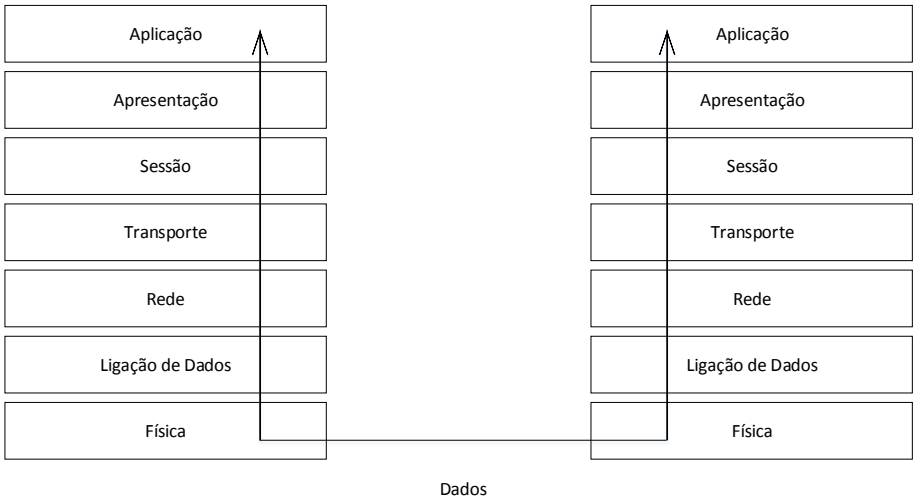

Figura 2.1: Modelo da pilha protocolar OSI.

- **Camada de transporte** - camada responsável pelo transporte dos bits até ao destino. Por exemplo, *Transmission Control Protocol* (TCP) e *User Datagram Protocol*(UDP);

- **Camada de sessão** - responsável pela gestão de sessões entre diferentes entidades. Por exemplo, *Remote Procedure Call* (RPC);

- **Camada de apresentação** - responsável pela representação, cifras e outras operações sobre os dados;

- **Camada da aplicação** - responsável pela passagem dos dados para a aplicação final à qual os dados se destinam. Por exemplo, *File Transfer Protocol* (FTP), *Simple Mail Transfer Protocol* (SMTP) e *Secure Shell* (SSH).

Este é o modelo existente na maioria das entidades mais comuns que compõem uma rede, mais precisamente nos terminais (computadores) comunicantes.

Os principais protocolos da camada de transporte da Internet são o TCP e o UDP. Ambos são utilizados para transportar informação entre duas entidades que comunicam entre si, embora tenham uma lógica de funcionamento distinta. Como primeiro objetivo, a Interface Díodo está preparada para receber dados que são transportados por estes dois protocolos.

O TCP é um protocolo de transporte de alta fiabilidade utilizado na comunicação entre entidades que pertencem a redes de comunicação baseadas em comutação de pacotes e sistemas interligados através dessas redes [9]. Entende-se por alta fiabilidade a garantia de que os pacotes enviados por um emissor serão entregues ao recetor. Denomina-se este tipo de garantia por ligação orientada ao estado. Este objetivo é conseguido através de um conjunto de campos no cabeçalho do TCP, permitindo às partes comunicantes a retransmissão de pacotes, o conhecimento por parte do emissor de que o pacote foi entregue, etc.

Para garantir o correto funcionamento do protocolo TCP, é necessário uma troca de mensagens em ambos os sentidos da ligação. O emissor dos dados envia não só os dados propriamente ditos mas também informação de fluxo, como SYN, ACK, FIN, entre outros; o recetor também envia mensagens de confirmação, retransmissão entre outros procedimentos. Por estas razões, diz-se que a lógica do protocolo TCP é bidirecional.

Um exemplo claro e demonstrativo da bidirecionalidade do protocolo TCP é o seu processo de inicialização. Como ilustra a figura 2.2, o início do processo de ligação entre emissor e recetor dá-se com um pedido SYN emitido pelo entidade que pretende iniciar a comunicação. O recetor responde com um $SYN + ACK$, significando que o recetor confirma e aceita o pedido de ligação. A processo dá-se por concluído quando o recetor recebe este $SYN + ACK$ e responde com um ACK. A partir deste momento o emissor pode enviar dados. Esta sequência de trocas de informação de controlo demonstra o sentido bidirecional do protocolo. Durante o período de uma ligação, desde o seu estabelecimento até ao seu término, estes acontecimentos bidirecionais mantêm-se, embora com outra sequência de respostas.

Em oposição ao protocolo TCP, o protocolo UDP fornece um procedimento para os programas aplicacionais trocarem mensagens com o mínimo de esforço. O protocolo não é orientado à ligação, ou seja, não existe informação para controlo de fluxo, e não é garantida a entrega nem a prevenção de duplicados. As aplicações que necessitem de entrega fiável ordenada deverão utilizar o TCP [10]. Por esta razão, a informação de cabeçalho do UDP é simplista e não contém qualquer informação de controlo. Este protocolo faz o menor esforço para transportar um pacote de um ponto da rede para o outro.

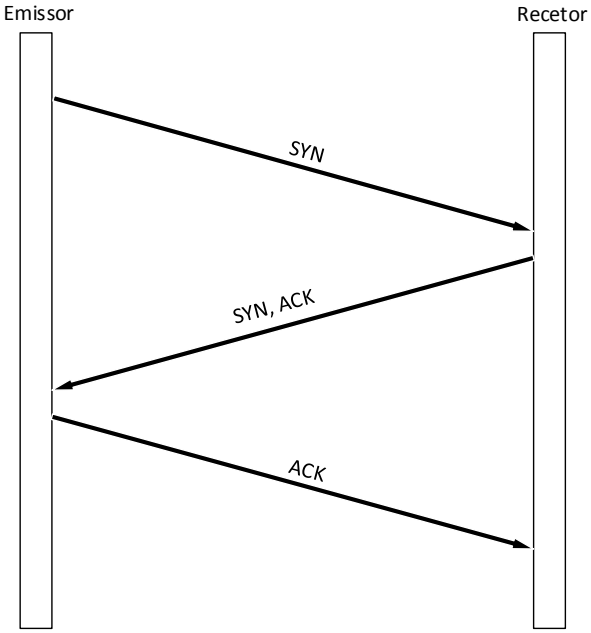

Figura 2.2: *3-way handshake* do TCP para o início de uma ligação entre duas entidades comunicantes.

Voltando ao conceito de díodo, denota-se claramente que o protocolo UDP se comporta de maneira idêntica à pretendida. Porém, a grande maioria das aplicações assentam sobre o protocolo TCP. É importante diferenciar a lógica do protocolo de transporte utilizado com a lógica aplicacional. Mesmo que a grande maioria das aplicações utilizem um protocolo bidirecional, a sua lógica aplicacional é unidirecional. Se nos focarmos nos termos aplicacionais do e-mail, este é enviado e não existe qualquer informação no sentido oposto. Se, por outro lado, na visita a uma página *web*, o utilizador envia um pedido e espera por resposta. Neste caso, estamos perante uma lógica aplicacional bidirecional.

No contexto do díodo, pretendemos tratar toda a informação que chega até ele como unidirecional. Por essa razão, o resultado deste trabalho tem um conjunto muito bem definido de utilizações. Por exemplo, não faz sentido utilizar a Interface Díodo para pedidos HTTP (HTTP), uma vez que a maioria destes pedidos resulta numa resposta. Por outro lado, faz sentido utilizar a Interface Díodo para a transferência de um pacote de atualização de um sistema operativo proveniente de uma rede com um baixo nível de segurança. A questão que se levanta é como utilizar um dispositivo unidirecional quando a maioria das aplicações utiliza um protocolo de transporte bidirecional.

Estes são os dois protocolos de transporte suportados pela Interface Díodo. Desta forma, o tráfego que chega à Interface Díodo quer utilizando TCP quer utilizando UDP serão transmitidos pelo serviço até ao dispositivo final.

2.2 Tolerância a Faltas

Uma das contribuições deste trabalho é a realização de um díodo de dados tolerante a faltas. Entende-se por tolerância a faltas a capacidade de um dispositivo ou sistema sobreviver a comportamentos anormais. Nesta secção define-se o conceito de tolerância a faltas (por paragem) e tolerância a faltas Bizantinas.

2.2.1 Tolerância a Faltas por Paragem

A tolerância a faltas é um conceito que surgiu da nova abordagem necessária para enfrentar a massificação de serviços que passaram para o mundo online e o quão dependentes estamos deles. A indisponibilidade de um serviço fornecido por determinada empresa pode causar-lhe uma enorme despesa monetária bem como a perda de clientes. Por essa razão, é necessário construir sistemas que continuem a funcionar corretamente mesmo quando alguns dos seus constituintes assumem um comportamento incorreto. Enquanto as técnicas de prevenção de falhas trabalham antecipando a sua ocorrência, as técnicas de tolerância a faltas não trabalham com antecipação de falhas [11], mas sim com a possibilidade destas ocorrerem.

Sem um esquema de tolerância a faltas, quando um componente de um sistema falha, depois de quebrar todas as técnicas de prevenção, diz-se que falhou por *Fail-Stop*[1]. Isto significa que existe uma falta que gerou um erro e esse erro não foi devidamente tratado, levando o componente para um estado de paragem. As técnicas para tolerância a faltas são realizadas com base em redundância (varias instâncias/réplicas que executam o mesmo serviço), e além de oferecerem várias réplicas que prestam o mesmo serviço, contam ainda com deteção de erros e respetiva recuperação.

Do ponto de vista da redundância, quando um componente falha, podem ser

[1]Em tolerância a faltas assume-se que um componente falha por *Fail-Stop*, ou seja, cessa por completo o seu funcionamento

tomadas duas ações:

- Mascaramento - o sistema contínua a evoluir com normalidade mas sem ter em conta a réplica que falhou;
- Remoção - é detetada a falha da réplica e esta é reiniciada ou substituída por outra réplica.

2.2.2 Tolerância a Faltas Bizantinas

O conceito de Tolerância a Faltas Bizantinas deve o seu nome ao problema dos Generais Bizantinos exposto e popularizado por Lamport et al. [4]. Nesse problema, os generais do Império Bizantino, que se encontram separados geograficamente, têm que tomar uma decisão sobre se atacam ou não o inimigo, sem contatarem diretamente uns com os outros. A única forma de comunicação utilizada pelos generais é realizada através do envio de um mensageiro que fica responsável por transmitir uma mensagem (proposta). No entanto, nada garante que esses mensageiros e os próprios generais sejam fiéis; na verdade, podem ser traidores e transmitir uma mensagem diferente para cada um dos outros generais. Por exemplo, um general pode enviar um mensageiro com a proposta de atacar e enviar outro mensageiro com a mensagem para recolher. Um general também pode tomar uma decisão contrária àquela que lhe foi transmitida, como atacar quando todas as mensagens que recebeu foram de defender.

Transpondo o problema dos Generais Bizantinos para os sistema informáticos, o problema passa a incidir sobre os diversos componentes que o compõem. Assim sendo, a tolerância a faltas obtém-se aumentando o número de réplicas para determinado serviço e a tolerância a comportamentos Bizantinos obtém-se recorrendo a várias instância do mesmo serviço que podem substituir o componente erróneo, garantindo que o sistema continua a funcionar corretamente. Por outras palavras, o conceito de tolerância a faltas Bizantinas vai além da falha física da máquina (*Fail-Stop*). Um componente não só pode falhar por paragem, sendo uma falha no domínio do tempo, como pode falhar no domínio da semântica (falhas arbitrárias e por omissão), através de computação incorreta, que leva não só ao seu estado erróneo como poderá tentar que as outras componentes do sistema se tornem incoerentes. Esta ideia introduz uma nova necessidade na construção de sistemas que está relacionada com o número de réplicas ne-

cessárias para que a tolerância a faltas seja efetiva. O número de réplicas que executam o mesmo serviço está dependente do número de réplicas faltosas que o sistema suporta. Assim, introduzem-se duas equações (réplicas com e sem estado[2]) que definem esse número: $n = 2f + 1$ (com assinaturas) e $n = 3f + 1$ (sem assinaturas), sendo n o número de réplicas e f o número de faltas suportadas pelo sistema. A ideia por trás deste número é o mesmo em ambas as equações e serve para garantir que existe uma maioria de réplicas corretas.

2.3 Díodos de Dados

2.3.1 Díodo Ótico

O início da história do díodo de dados data de 1960, começando a ser desenvolvido e produzido para consumo comercial em 1999 [5]. Até há relativamente pouco tempo, os díodo de dados eram utilizados somente no contexto militar e só recentemente começaram a ser utilizados noutros contextos. A solução comercial, o díodo de Dados Ótico [5], é composto por dois dispositivos de fibra ótica, cada um dotado de um canal de envio e receção. Ligando apenas o canal emissor de um dispositivo ao recetor do outro, mas não o contrário, garante-se uma comunicação unidirecional. Além da primeira concretização, têm sido desenvolvidas outras soluções do díodo de dados [12, 13], que também têm como base a utilização de dispositivos de fibra ótica interligados por um cabo de fibra ótica. O fluxo de informação dentro do díodo é transportado através do protocolo UDP. A figura 2.3 representa a concretização do fluxo unidirecional do tráfego num díodo de dados que medeia a comunicação entre rede insegura e rede segura.

Cada dispositivo é composto por duas portas, uma para transmissão e outra para receção de dados. É realizada uma ligação entre a porta de transmissão do primeiro dispositivo e a porta de receção do segundo dispositivo. Não existe qualquer ligação entre a porta de transmissão do segundo dispositivo e a porta de receção do primeiro, impedindo a transmissão de dados nesse sentido. Não

[2]Entende-se por estado a informação de sessão guardada por uma réplica durante um determinado período de tempo. Se uma réplica não mantém guardada nenhuma informação de sessão (e.g. *stateless firewall*), então é uma réplica sem estado. Por outro lado, se uma réplica guarda informação (e.g. *cache*), considera-se uma réplica com estado.

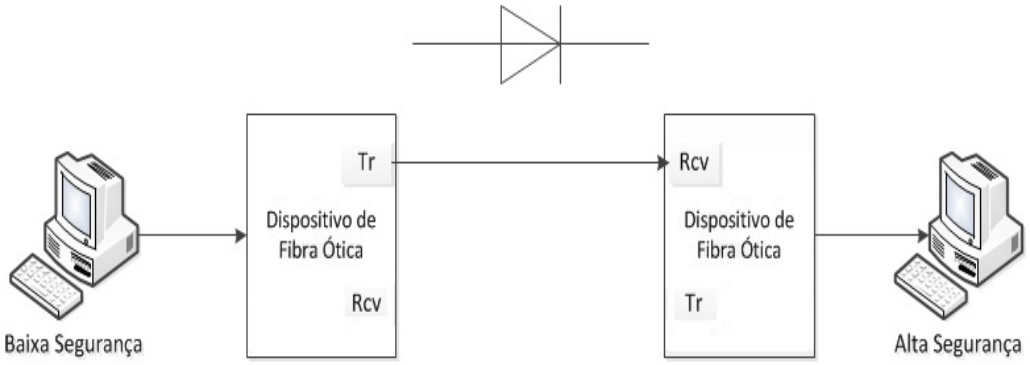

Figura 2.3: Díodo de Dados Ótico.

existindo ligação física entre a porta de emissão do dispositivo da rede segura e a porta de receção do dispositivo da rede insegura, o díodo garante que não existe tráfego nesse sentido. Nesta concretização, o tráfego que chega ao díodo de dados é transportado pelo protocolo UDP. Uma vez no díodo, os dispositivos traduzem o protocolo unidirecional para protocolos bidirecionais *standard* [14] (como o TCP).

O díodo de dados tem como objetivo garantir duas propriedades de segurança, a confidencialidade[3] e a integridade[4] dos dados mantidos no dispositivo que se encontra na rede de alto nível de segurança (ANS).

A confidencialidade é garantida no modo de operação baixo nível de segurança (BNS) para ANS, onde são enviados dados de um domínio de segurança inferior para um domínio de segurança superior, não sendo possível ler os dados que se encontram no dispositivo na rede de alta segurança (recetor).

A integridade dos dados no ANS pode ser violada caso seja possível que um utilizador malicioso no BNS modifique informação no ANS quando a entidade presente nesta última rede assume o papel de transmissor (ligação contrária à da figura 2.3). Uma vez que apenas saem dados do dispositivo no ANS, não é possível modificá-los a partir de um BNS.

[3]Confidencialidade é a medida em que um serviço ou informação está protegida contra a divulgação não autorizada.

[4]Integridade é a medida em que um serviço ou informação está protegida contra modificações não autorizadas.

2.3.2 *Pump*

Um sistema de segurança multi-nível guarda e processa informação com diferentes graus de sensibilidade. Isto significa que, de fato, existe um controlo que impede a passagem de informação proveniente de um ANS para um BNS.

É sabido que os requisitos de fiabilidade na entrega de mensagens passam pela utilização de um sistema de *acknowledgments*, como é o caso do protocolo de transporte TCP. Estes fluxos de *ACK* introduzem canais de comunicação dissimulados, e o próprio paradigma criado por estes protocolos, onde o domínio superior envia confirmações de receção de mensagens para o domínio inferior, viola o propósito do sistema de segurança multi-nível. O *Pump* é uma solução para o envio seguro e fiável de mensagens entre um BNS e um ANS, ao mesmo tempo que minimiza a ameaça subjacente às mensagens de confirmação *ACK* sem penalizar o desempenho e a fiabilidade do sistema. É o mesmo que utilizar o protocolo UDP, que não oferece garantias de entrega fiável mas com uma probabilidade de entrega associada que é representada por confirmações *ACK* estocásticas.

Existem duas concretizações do *Pump*: *Pump* básico e *Pump* de rede [6, 7]. O *Pump* básico é utilizado para servir apenas um emissor e um recetor. O *Pump* de rede serve vários emissores e recetores de diferentes aplicações.

Pump Básico

O *Pump* básico é um dispositivo de rede simples o suficiente para facilitar a sua avaliação, fiável e com um número reduzido de canais dissimulados. Um canal dissimulado, introduzido por B.W. Lampson em 1973, é um canal utilizado para troca de informação de sessão que por norma não é utilizado para comunicação. Tipicamente, estes canais existem para utilização em protocolos que requerem algum tipo de acordo antes da transmissão de dados. Embora estes canais violem o princípio da unidirecionalidade, se essa informação for limitada e muito bem definida não compromete o correto funcionamento do díodo.

A figura 2.4 representa uma visão abstrata do *Pump* básico. O *Pump* é constituído por um *buffer* não volátil que se encontra entre os dois domínios de rede. É responsabilidade do *Pump* enviar confirmações para o BNS de forma estocástica. Este envio probabilista é baseado no tempo entre o envio de dados do *buffer*

para o ANS e o tempo que esse domínio leva a responder uma confirmação de volta para o *Pump*. Através do envio estocástico destas confirmações para o BNS e com base na taxa de resposta do domínio de alta segurança, o *Pump* básico fornece entrega fiável sem penalizar o desempenho.

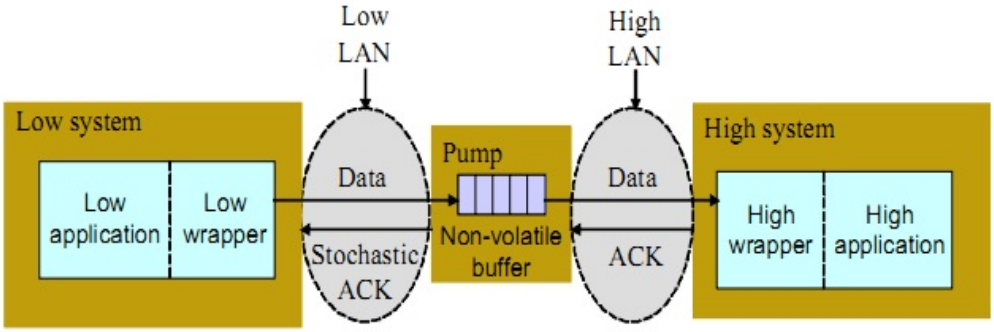

Figura 2.4: *Pump* básico.
[6]

Em cada extremo da comunicação existe um *software* que é responsável por comunicar com o *Pump* através de uma *Local Area Network* (LAN). Esse *software* é denominado por *wrapper*. Cada *wrapper* é composto por duas partes: uma específica do *Pump* e outra específica da aplicação. A primeira consiste numa biblioteca de rotinas que concretizam o protocolo *Pump*; a segunda tem mecanismos para invocar as rotinas oferecidas pela *Appliation Programming Interface* (API) do protocolo *Pump*. Como ambos os protocolos são do nível aplicacional, o *Pump* oferece fiabilidade entre aplicações.

Esta estrutura de *wrappers* tem duas vantagens:

- A confidencialidade do *Pump* depende apenas dele próprio e não dos *wrappers*. Assim sendo, o *software* que concretiza os *wrappers* não é crítico para o sistema;

- Os *wrappers* tornam o *Pump* num dispositivo genérico que é independente de uma aplicação em específico. Desta forma o *Pump* pode ser utilizado em conjunto com várias aplicações.

Pump de Rede

O *Pump* básico é utilizado para comunicações entre um emissor e um recetor. O *Pump* de rede atua como um *router* que interliga aplicações de um nível inferior de segurança a aplicações de um nível superior de segurança. Em relação ao *Pump* básico, esta concretização oferece as mesmas propriedades e acrescenta mais duas: justiça e prevenção a ataques de Negação de Serviço (DoS).

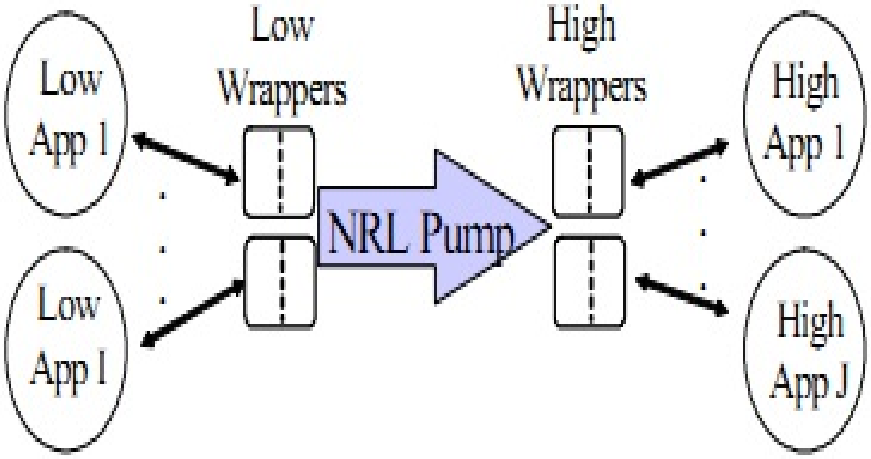

Figura 2.5: *Pump* de rede.
[6]

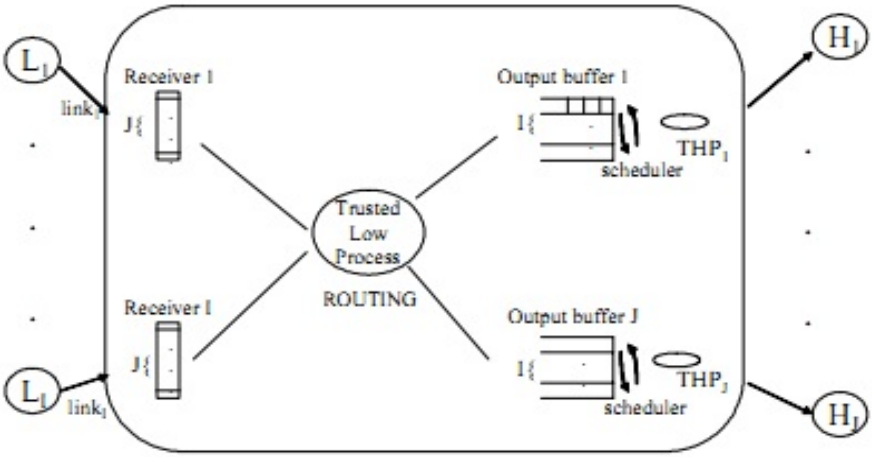

Figura 2.6: Mecanismo interno do *Pump* de rede.
[6]

As figuras 2.5 e 2.6 apresentam a estrutura global do *Pump* de rede. O funcionamento do sistema para garantias de justiça e prevenção a ataques de (DoS) está estruturado, como proposto em [6], da seguinte maneira: L_i e H_j (L - *Low* e H - *High*) são o conjunto de dados que entram e saem, respetivamente, na rede do *Pump*. As entradas (e saídas) consistem em processos i (j) que não comunicam entre si. Cada L_i pode enviar mensagens para qualquer H_j. Considerando a $sessao_{ij}$, após L_i enviar uma mensagem para H_j, o primeiro fica à espera do *ACK* enviado pelo *Pump* de rede. Depois de recebido o *ACK*, L_i pode enviar outra mensagem para H_j. Isto significa que cada L_i só pode enviar uma nova mensagem em cada sessão depois de receber o *ACK* do *Pump de rede* referente à mensagem anterior. Quando H_j recebe uma mensagem do *Pump* envia um *ACK* de volta para trás.

Para cada L_i existe um recetor, um *buffer* com J *slots* onde a $slot_j$ guarda as mensagens da $sessao_{ij}$ até ser reencaminhada pelo *Trusted Low Process* (TLP). O TLP reencaminha a mensagem do recetor para o *buffer* de saída indicado. Existem I *buffers* lógicos de saída para H_j, cada um denotado como $buffer_{ij}$. Uma mensagem da $sessao_{ij}$ será guardada no $buffer_{ij}$. Posteriormente, o *Trusted High Processes*(THP), THP_j entrega uma mensagem do $buffer_{ij}$ a H_j de acordo com o esquema de escalonamento. O THP_j não pode entregar mais mensagens do $buffer_{ij}$ até ser enviado o *ACK* por H_j referente à *primeira mensagem do* $buffer_{ij}$.

Além destas duas soluções (díodo de dados ótico e *pump*), existe ainda o díodo de dados iterativo. Esta concretização difere das anteriores pelo fato de aceitar como dados de entrada pacotes transportados tanto por TCP como por UDP.

A concretização desta solução pode ser concretizada tanto com fibra ótica como com ligações série (RS-232) [14].

2.4 Limitações das Soluções Atuais

O díodo de dados apresenta um conjunto de desvantagens para a atual realidade. Uma das desvantagens é a falta de redundância da sua concretização. Se um dos dispositivos de fibra ótica falhar, todo o sistema falha. Além desta possibilidade de falha por *Fail-Stop*, o díodo é composto por uma componente de *hardware* (os dois dispositivos e o cabo de fibra ótica) e componentes de *software*,

que providenciam o mecanismo para a comunicação através do canal, como por exemplo a projeção dos pacotes que entram para as entidades finais. Sendo *software*, a probabilidade de existência de vulnerabilidades aumenta e, por isso, o díodo sofre do problema de que se um dos seus dispositivos for dominado por um utilizador malicioso, o correto funcionamento do mesmo fica comprometido porque este componente é um ponto singular de falha.

Outra desvantagem é a consequência de utilização de fibra ótica. O cabo de fibra ótica que liga os dois dispositivos sofre de reflexões. Este fenómeno é causado pelo *air gap* utilizado na fibra ótica, que não é mais do que uma junção utilizada para ligar fibra ótica. Quando é utilizado, a luz emitida dentro do cabo de fibra ótica pode ser refletida para trás, fenómeno denominado por *backreflection* [5] (BR) Estas reflexões são dados refletidos para trás, violando o conceito de díodo.

O díodo de dados utiliza apenas o UDP como protocolo de transporte. Na proposta de Hamed Okhravi et al. [14] é dada uma visão sobre a utilidade dos díodos de dados para obter confiabilidade em infraestruturas industriais, salientando algumas das limitações deste dispositivo de rede. Essas limitações estão relacionadas com a impossibilidade de utilização do protocolo de transporte TCP, uma vez que a utilização deste protocolo quebra o conceito de díodo. Esta afirmação pode ser refutada dependendo da interpretação do conceito de díodo. Se a informação trocada no sentido oposto for somente de controlo, então as propriedades de segurança (confidencialidade e integridade) continuam a ser garantidas. Por outro lado, recorrendo ao UDP como protocolo de transporte, embora não haja informação de controlo no sentido oposto da ligação, não existe qualquer garantia de entrega de dados entre as duas entidades.

Por último, algumas das concretizações faladas anteriormente (por exemplo, o díodo de dados interativo) necessitam de um díodo por cada serviço fornecido. Assim sendo, o número de díodos necessário é igual ao número de serviços fornecidos pela rede de ANS. Sempre que se pretende adicionar um serviço é necessário adicionar um novo díodo para mediar a comunicação.

[5] http://www.kingfisherfiber.com/Application-Notes/06-Optical-Return-Loss-Testing.htm

2.5 Sumário

Existem várias realizações do díodo de dados sendo todas elas maioritariamente baseadas em *hardware*. É o que advogam as empresas que os fabricam. No entanto, existe uma componente de *software*, que é um ponto singular de falha, responsável por tratar os dados nos dispositivos de fibra ótica.

O díodo de dados ótico é uma realização do díodo de dados e não utiliza qualquer informação de controlo. Isto significa que só aceita dados transportados por UDP. O *Pump* é outra concretização de díodo que utiliza um esquema de probabilidades para gerar alguma informação de controlo. Existem ainda os díodo de dados interativos que aceitam dados transportados por TCP ou UDP.

Capítulo 3

Interface Díodo

Este capítulo contém uma descrição arquitetural da Interface Díodo. É composto por uma descrição do modelo do sistema e seus pressupostos, e uma descrição da arquitetura, bem como das tecnologias que compõem os componentes internos.

É essencial manter as propriedades de segurança dos dados que se encontram em redes com alto nível de segurança (ANS) quando os componentes nelas inseridos comunicam ou fornecem um serviço a entidades que se encontram num rede com um nível de segurança inferior. A Interface Díodo providencia uma comunicação unidirecional [12, 7, 6, 13, 14, 5] para o transporte de mensagens enviadas por um cliente numa rede insegura para uma entidade numa rede segura.

3.1 Modelo do Sistema

3.1.1 Modelo de Rede

Assumimos que a rede é totalmente ligada e utiliza o modelo TCP/IP. Isto significa que tanto clientes como entidades finais conseguem comunicar com a Interface Díodo. O protocolo de transporte utilizado na comunicação entre entidades e a Interface Díodo é *Transmission Control Protocol*(TCP) ou *User Datagram Protocol* (UDP). As redes virtuais internas ao díodo conseguem comunicar dentro do ambiente virtual e não têm acesso ao exterior.

3.1.2 Modelo de Faltas

Assumimos que todos os componentes não replicados são seguros, ou seja, não têm comportamentos Bizantinos e só falham por paragem (*Fail-Stop)*. Este pressuposto deve-se ao fato de serem componentes com um baixo grau de complexidade, compostos por tecnologias bem conhecidas e utilizadas em grande escala. Isto significa que as suas vulnerabilidades, quando existem, são bem conhecidas e rapidamente corrigidas.

Devido à sua complexidade, consideramos a possibilidade de existência de vulnerabilidades nos componentes replicados da Interface Díodo, como ataques ao sistema operativo devido a pacotes mal formados, problemas associados à própria linguagem de concretização, etc. Como tal, oferecemos resiliência a diferentes tipos de faltas. Em concreto, assumimos dois tipos de faltas: faltas acidentais (como uma réplica ir a baixo) e faltas maliciosas (vulnerabilidades exploradas com o objetivo de danificar o correto funcionamento do sistema, tais como modificação e omissão de mensagens). Relativamente às faltas acidentais, são toleradas f faltas acidentais para o número de réplicas igual a $3f + 1$. No segundo caso, o sistema pode tolerar f faltas Bizantinas para $3f + 1$ réplicas.

Assumimos o ambiente virtual como seguro. A correção da camada virtual fica a cargo do *hypervisor* (o seu posicionamento da concretização da Interface Díodo é demonstrado na figura 3.3) e este não pode ser comprometido.

Não é feito qualquer pressuposto sobre o comportamento dos utilizadores (clientes). Os dados enviados por um cliente (corretos ou incorretos) serão entregues à entidade final mesmo perante a presença de faltas.

3.2 Arquitetura

A Interface Díodo pode ser decomposta em três camadas de comunicação, como apresentado na figura 3.1 e 3.2. A primeira camada é a interface de comunicação de entrada e saída, que providencia as interações básicas entre utilizadores e o serviço. Este componente não é replicado e só falha por paragem. A segunda camada é composta por várias réplicas, os gestores de projeções, que transformam os pacotes enviados por um utilizador num novo tipo de pacotes de forma a serem reencaminhados para as réplicas finais. Estas réplicas contêm a mesma

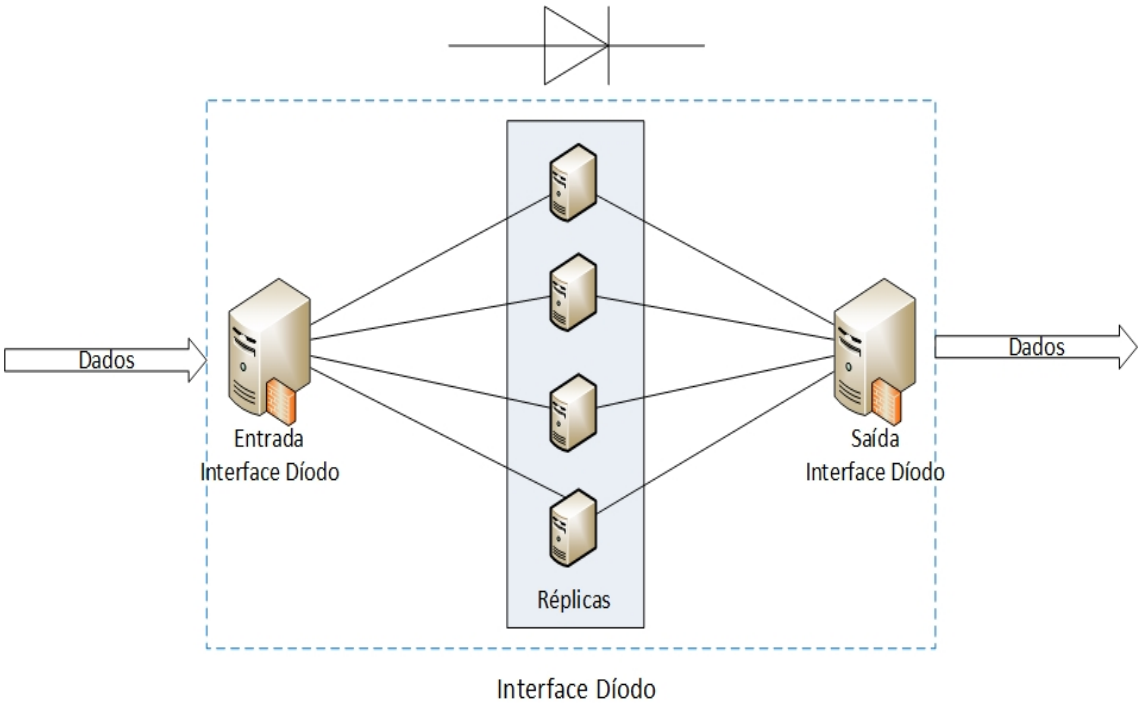

Figura 3.1: Arquitetura da Interface Díodo.

pilha de *software*, podem ser Bizantinas e falhar maliciosa ou acidentalmente. A terceira camada é o monitor, que monitoriza o comportamento da Interface Díodo e é uma entidade simples e com baixo grau de complexidade. Este monitor identifica comportamentos incorretos tais como:

- Gestores de projeção enviam dados no sentido inverso do que era suposto;
- Réplicas maliciosas a enviar uma quantidade de dados acima ou abaixo da taxa de envio das outras réplicas. Este comportamento pode ser qualificado como um ataque de Negação de Serviço (*Denial of Service*, DoS), por ser uma tentativa de inundar a entidade final com um número de pedidos com os quais esta não pode lidar.

Os pacotes criados pelos gestores de projeção são objetos próprios da Interface Díodo e, dos pacotes originais enviados pelos utilizadores, só se retiram os dados. Os novos pacotes contêm o endereço e porto do serviço da máquina final, um identificador do pacote e os dados originais.

O serviço oferecido pela Interface Díodo pode ser utilizado por vários clientes em simultâneo para diferentes serviços finais. Por exemplo, podemos ter um

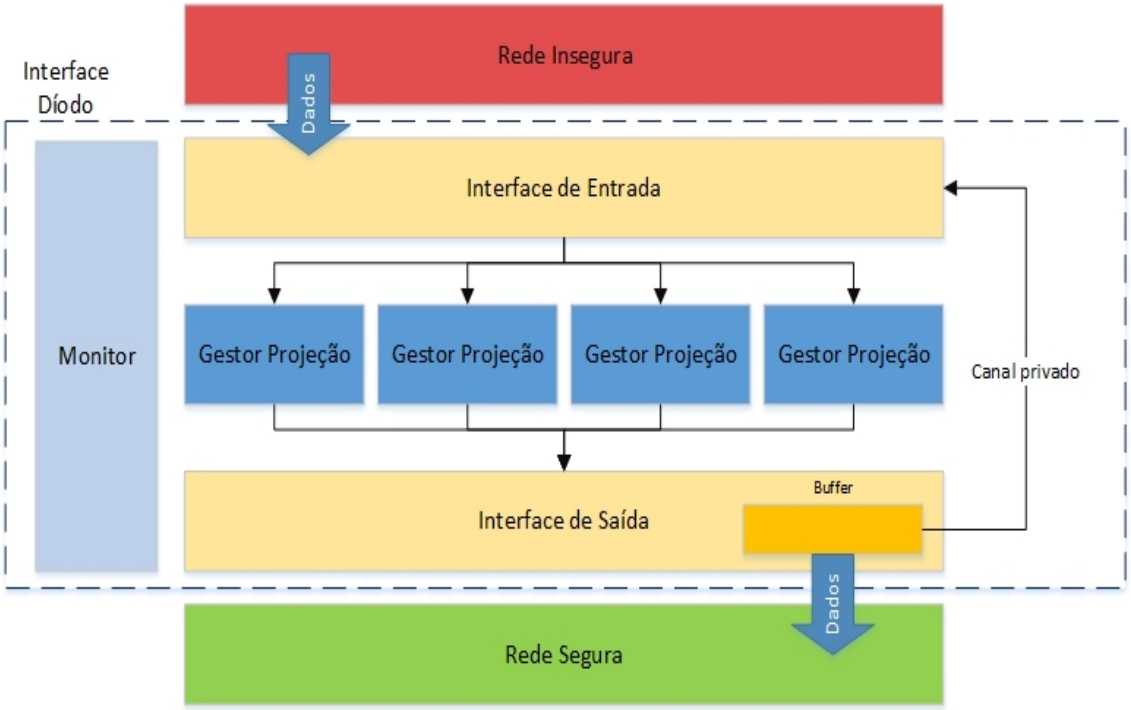

Figura 3.2: Arquitetura pormenorizada da Interface Díodo.

gerador de *logs* a transmiti-los para uma entidade final e, ao mesmo tempo, um servidor de e-mail a enviar e-mails para um servidor de *backup*. Os dados são criados e enviados por cada cliente e são entregues à interface de entrada do díodo. Uma vez aí, são realizadas uma série de operações sobre os pacotes dentro da Interface Díodo e, posteriormente, a interface de saída entrega os dados ao serviço correspondente. Quando existem dados a fluir no sentido contrário ao funcionamento da Interface Díodo, as entidades de entrada e saída atuam também como bloqueadoras de tráfego.

Por razões de controlo de fluxo, é necessário que a interface de saída comunique com a de entrada. Essa comunicação é realizada através de um canal privado e serve para controlar a qualidade do serviço, como por exemplo, quando a velocidade de entrada de dados é substancialmente superior à de saída. Neste caso, a interface de saída comunica com a de entrada para que esta proceda a um abrandamento.

Todos os componentes que constituem o serviço são virtuais. Isto significa que o sistema é composto por um ambiente virtual. Na figura 3.3 está uma representação da arquitetura do ambiente virtual. Os componentes virtuais co-

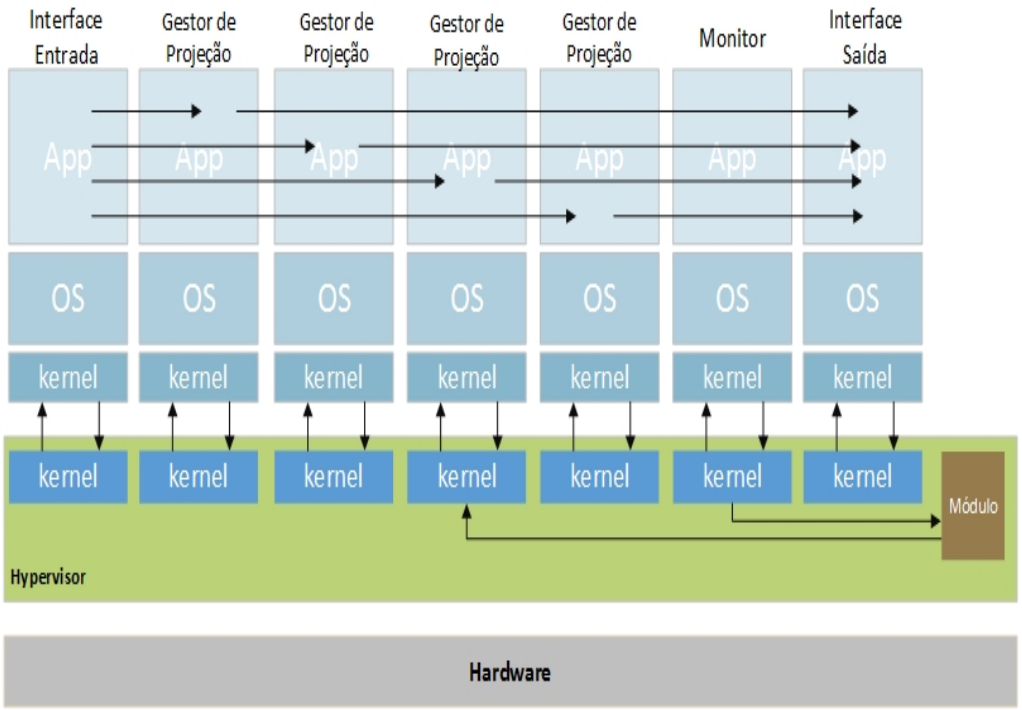

Figura 3.3: Arquitetura do ambiente virtual da Interface Díodo.

municam através de redes virtuais. Como a figura demonstra, a interface de entrada envia as mensagens recebidas para todos os gestores de projeção e estes enviam a sua mensagem para a interface de saída. O monitor, em caso de deteção de alguma anomalia, comunica com um módulo no *hypervisor* sobre estes comportamentos este emite um sinal ao *kernel* da máquina virtual que questão para que esta seja reinicializada. Uma rede de computadores é uma ligação de comunicação entre dois ou mais dispositivos. As redes podem ser físicas ou virtuais. Se a ligação é realizada através de um cabo, dispositivo *wireless* ou *bluetooth*, a rede é física. Se a comunicação é realizada entre várias máquinas a executar no mesmo dispositivo físico mas num ambiente virtual, é uma rede virtual. O modelo de comunicação utilizado é híbrido: redes físicas entre entidades que comunicam com a interface de entrada e saída e redes virtuais dentro da Interface Díodo. Existe uma rede virtual entre a interface de entrada e os gestores de projeção, entre os gestores de projeção e a interface de saída e um canal escondido entre as interfaces de entrada e a saída.

Idealmente, cada réplica executa um sistema operativo distinto. Desta forma obtém-se diversidade de sistemas operativos [15, 16], diminuindo a probabilidade de acontecer um ataque executado da mesma maneira duas vezes segui-

das. Através da diversidade, e sabendo de antemão que não é possível assegurar a inexistência de vulnerabilidades nos sistemas operativos, a probabilidade de duas máquinas virtuais terem as mesmas vulnerabilidades diminui.

3.3 Tecnologias

Nesta secção são apresentadas e descritas as tecnologias utilizadas para a concretização da Interface Díodo. A concretização da Interface Díodo utiliza maioritariamente tecnologias já desenvolvidas e bem conhecidas combinadas numa arquitetura inovadora. As interfaces de entrada e saída, como ponto de contato entre as duas redes, executam uma *firewall* iptables. As réplicas de gestão de projeção, através de uma ferramenta denominada por JPCap, capturam os pacotes enviados pelo utilizador. O monitor, que verifica o comportamento do Díodo, é um *Network Intrusion Detection System* (NIDS) realizado utilizando snort.

3.3.1 Replicação

A replicação é um mecanismo cooperação entre entidades redundantes, que tem como objetivo a coerência dos dados, tolerância a faltas e disponibilidade, quer em *hardware* como *software*. A replicação pode ser física ou virtual. Na replicação física, determinado serviço é replicado através de vários componentes físicos que executam o mesmo programa. Com a replicação virtual, um dispositivo físico é suficiente, uma vez que as réplicas são virtuais, ou seja, partilham os mesmos recursos. Na Interface Díodo utilizamos a virtualização como meio de concretização de um esquema de replicação.

Além dos recursos utilizados, existem dois tipos de replicação:

- Replicação de Dados - utilizada para replicar dados por diversos componentes. Desta forma, caso um dos componentes falhe, existem outros componentes com os mesmos dados.
- Replicação de Computação - utilizada para replicar computação, ou seja, diversos componentes replicados executam exatamente as mesmas operações. Este tipo de replicação é denominado por Replicação de Máquina de Estados [17].

Neste trabalho, o foco inside sobre a replicação da computação. Isto significa que em determinada parte da arquitetura da Interface Díodo existem componentes replicados para executarem exatamente a mesma computação, pelo menos enquanto apresentam um comportamento correto. Este esquema de replicação tem três aspetos chave para a obtenção de tolerância a faltas:

- **Coordenação** - todas as réplicas recebem e processam a mesma sequência de pedidos;
- **Acordo** - todas as réplicas corretas recebem todos os pedidos;
- **Ordenação** - todas as réplicas corretas processam os pedidos recebidos pela mesma ordem.

Assim sendo, a Replicação de Máquinas de Estado parte do pressuposto que se duas réplicas do mesmo serviço, a e b, partem de um estado x e chegam a um estado y, então $y_a = y_b$.

A replicação tem um conjunto de vantagens associado à sua utilização:

- **Diversidade** - permite a realização de um esquema de diversidade [15, 16]. Cada réplica pode ter um sistema operativo distinto das demais réplicas, dificultando a vida a um utilizador malicioso uma vez que a probabilidade de existir a mesma vulnerabilidade em dois sistema operativos diferentes é reduzida ;
- **Redundância** - possibilita redundância, ou seja, caso uma réplica constituinte do sistema falhe, existe uma réplica que a poderá substituir. Desta forma, o sistema continua operacional (disponibilidade);
- **Recuperação** - vários mecanismos e protocolos assentam sobre ou recorrem à replicação. Por exemplo, a recuperação pro-ativa só é possível porque existem $n - k$ máquinas num determinado instante, onde n é o número de máquinas e k o número de máquinas a recuperar, existindo n máquinas a realizar o mesmo trabalho que as k máquinas em recuperação.

3.3.2 Virtualização

A virtualização é um mecanismo para obter replicação e assume um papel preponderante no panorama geral dos sistemas atuais e, inclusive, no trabalho que

apresentamos. Essencialmente, porque se o ambiente virtual for comprometido, a Interface Díodo é comprometida.

Numa visão de alto nível, a virtualização consiste em várias instâncias do mesmo serviço a executar no mesmo dispositivo físico. Através deste paradigma, é possível uma otimização de recursos, maximizando o poder computacional dos componentes de *hardware* utilizados. Se por um lado a flexibilidade oferecida pelos ambientes virtuais é uma vantagem para o aproveitamento de recursos, a virtualização é alvo de atenções em termos de segurança.

A figura 3.4 apresenta a estrutura típica de um ambiente virtual. Um esquema de virtualização resume-se a quatro camadas principais: *hardware*, *hypervisor*, *Virtual Machine Monitor* (VMM) e os *guest Operating System* (Guest OS).

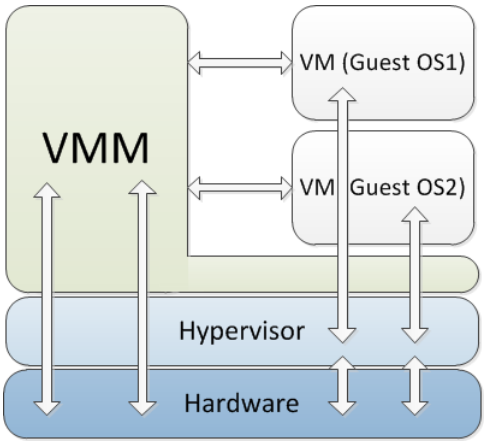

Figura 3.4: Pilha de camadas de um ambiente virtual. [18, 19, 20]

Seguindo a pilha de camadas do ambiente virtual, cada camada tem as seguintes tarefas:

- *Hardware* - responsável por oferecer as funcionalidades e serviços inerentes a um computador (como memória, armazenamento, placa de rede, etc.);
- *Hypervisor* - responsável pela comunicação entre as camadas acima e a camada de *hardware*, sendo por essa razão a camada de maior importância num ambiente virtual e necessita de especial atenção em termos de segurança;
- VMM - responsável por monitorizar as máquinas virtuais em execução. Um VMM não é mais do que um *hypervisor* com um nível de acesso inferior,

ou seja, é executado sobre um *host OS*[1];

- *Guest OS* - instâncias de máquina virtual e são os sistemas operativos utilizados diretamente pelo utilizador.

A utilização de um ambiente virtual para a concretização da replicação tem vantagens associadas [21]:

- Adaptação - como as máquinas virtuais se encontram na camada de *software*, a sua modificação torna-se mais fácil e efetiva quando comparada com a modificação física;
- Otimização - melhor aproveitamento dos recursos e capacidades do dispositivo físico. Ao executar várias réplicas num ambiente virtual, o aproveitamento dos recursos é superior quando comparado à replicação utilizando vários componentes físicos;
- Independência - uma vez que os *guest OS* são totalmente independentes do VMM e do *hypervisor*, a sua portabilidade é facilmente conseguida;
- Separação de processos - a virtualização permite uma clara separação entre os processos do *host OS* e os serviços virtuais fornecidos, incluindo os processos do *guest OS*. Em termos de segurança esta separação é útil porque os processos aplicacionais deixam de ser confiáveis para a máquina física e passam a ter confiança somente da máquina virtual;

Apesar das várias vantagens associadas à virtualização, são necessários esforços para manter o ambiente virtual seguro mesmo com o comprometimento do seu elemento chave, o *hypervisor*. Embora a Interface Díodo assuma o *hypervisor* como seguro, existem trabalhos que têm explorado medidas de segurança para manter as propriedades de segurança (tais como integridade e confidencialidade) do ambiente virtual mesmo perante o comprometimento desta camada fulcral. No HyperSafe [22] garante-se a integridade do *hypervisor* perante o seu comprometimento. Para isso, é utilizado um *non-bypassable memory lockdown*, onde se mantém informação referente ao *hypervisor* em páginas de memória que necessitam de ser desbloqueadas para serem modificadas. Desta forma, embora o *hypervisor* possa ser modificado, estas páginas em memória garantem que é

[1]Um *host OS* é o sistema operativo base e instalado na máquina física

possível restabelecer a versão original. No NoHype [23], é proposto um sistema que assenta em quatro ideias essenciais e que retira ao *hypervisor* a responsabilidade de alocar recursos dinamicamente, simular serviços de I/O (*input* e *output*) e endereçar chamadas ao sistema. As quatro ideias fundamentais deste sistema são:

(1) pré-alocar recursos, como processador e memória;

(2) utilizar dispositivos de I/O virtualizados;

(3) pequenas modificações no *guest OS* para realizar uma descoberta do sistema no momento de arranque;

(4) evitar indireções, trazendo a máquina virtual para um contacto direto com a camada de *hardware* subjacente.

3.3.3 JPCap

O JPCap é uma biblioteca Java *open source* para a captura e envio de pacotes. Nos sistemas operativos UNIX, o JPCap recorre à biblioteca LibPcap e em Windows à biblioteca WinPcap. Na Interface Díodo, o JPCap atua nos gestores de projeção e tem como objetivo capturar os pacotes reencaminhados pela interface de entrada.

Através da utilização de uma API simples, o JPCap permite capturar pacotes, ler e guardar a informação neles contida. Esta captura permite que sejam criados novos pacotes específicos à Interface Díodo, como descrito no capítulo 4. Desta forma, as *firewalls* de entrada reencaminham o tráfego para os gestores de projeção e é possível manipular os pacotes que chegam até estes sem manipulação de *sockets*.

3.3.4 iptables

Uma *firewall* é um componente de rede, concretizado tanto em *software* como em *hardware*, que tem como objetivo permitir ou bloquear a passagem de determinado conjunto de dados. Desta forma, a *firewall* concretiza políticas de segurança para uma determinada rede. As *firewalls* básicas são baseadas e construídas através de uma cadeia de regras. Estas regras definem que informação é aceite ou

barrada. A lista de regras é percorrida do início ao fim até que exista um padrão entre uma regra e determinado pacote. Caso o pacote não encaixe ou corresponda a um padrão de qualquer uma das regras, a *firewall* pode ser configurada para aceitar ou rejeitar por omissão.

As *firewalls* dividem-se em dois grupos principais [24]:

- Filtro de Pacotes - o tráfego passa através da *firewall* para os serviços finais de uma rede interna, e o seu conteúdo é analisado por filtros para decisão do tipo *passa* ou *não passa*.
- Proxies - o fluxo de tráfego começa ou acaba na *firewall*, sendo intercetado e processado por representativos dos serviços finais na rede interna.

O iptables [25, 26] é uma concretização de *firewall* em *software* para sistemas UNIX. Na Interface Díodo o reencaminhamento de pacotes e o bloqueio da passagem de informação no sentido oposto do funcionamento do díodo é realizado por iptables. Esta concretização divide-se em duas funções básicas:

- Reencaminhamento - a *firewall* é o ponto de contacto entre a rede emissora (segura ou insegura, dependendo do modo de funcionamento do díodo) e o díodo. Quando o iptables recebe um pacote, reencaminha-o para o componente seguinte (réplicas centrais, JPCap). Se o pacote for direcionado para um endereço IP não existente no protocolo de projeção de endereço, o pacote é rejeitado.
- Filtro de Pacotes - ambos os extremos do díodo executam uma *firewall* iptables que decide sobre a passagem dos conteúdos. Na tentativa de passagem de tráfego no sentido oposto ao funcionamento do díodo, o iptables rejeita a passagem dos pacotes.

O iptables é constituído por tabelas. Cada tabela tem uma cadeia de regras. Mais precisamente, existem quatro tabelas: filtro, NAT, raw e mangle. A tabela de filtro, NAT e raw são utilizadas na Interface Díodo. A figura 3.5 demonstra como um pacote é processado quando chega à *firewall* iptables. Podem ser tomadas várias medidas sobre um pacote dependendo das regras criadas para cada tabela. As próprias tabelas têm ordens de execução, por exemplo a tabela mangle é consultada primeiro que a tabela NAT.

As tabelas têm as seguintes caraterísticas e cadeias:

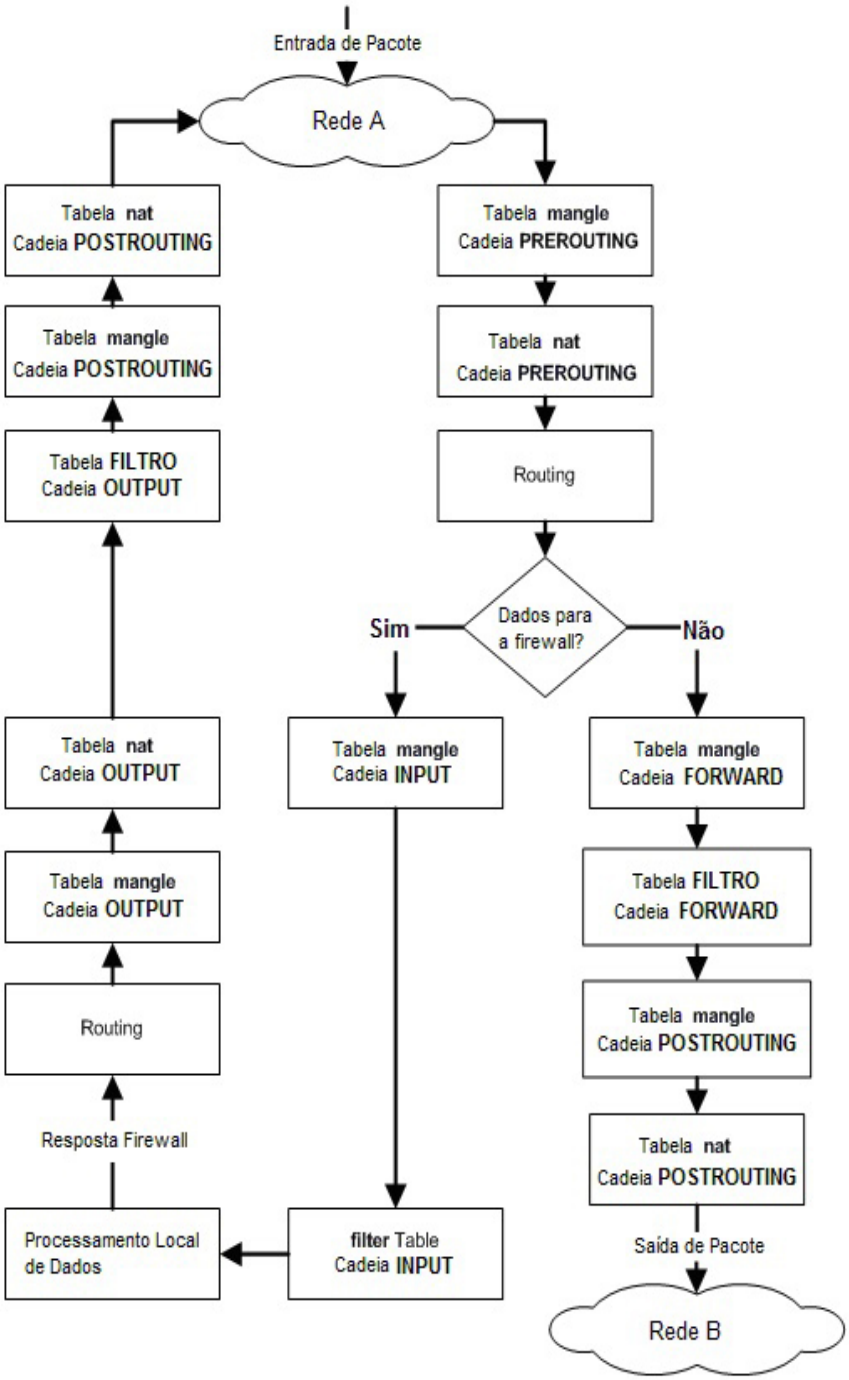

Figura 3.5: processamento de pacote no iptables.

- NAT - a tabela de NAT é responsável por receber pedidos externos e tratá-los para um endereço privado.

  – PREROUTING - as regras desta cadeia modificam os pedidos antes destes serem reencaminhados (i.e. a tradução do pacote acontece ime-

diatamente após dar entrada no sistema).

- POSTROUTING - as regras desta cadeia modificam os pedidos depois destes serem reencaminhados (i.e. a tradução do pacote acontece depois deste deixar o sistema).
- OUTPUT - as regras desta cadeia são utilizadas para pacotes gerados pela própria *firewall*.

- filtro - esta é a tabela por defeito do iptables
  - INPUT - as regras desta cadeia tratam os pacotes que entram na *firewall* (e.g. dados enviados por uma entidade numa rede externa).
  - OUTPUT - as regras desta cadeia tratam os pacotes que saem da *firewall* (e.g. resposta da *firewall* numa ligação TCP).
  - FORWARD - as regras desta tabela tratam e reencaminham os pacotes que chegam à *firewall* mas não são destinados para ela.

- raw - esta tabela é principalmente utilizada para colocar uma marca nos pacotes indicando que estes não devem ser tratados pelo sistema de ligação do Netfilter [2]. Este sistema permite ao *kernel* rastrear as todas as ligações lógicas de rede (ou sessões).
  - PREROUTING - as regras desta cadeia modificam os pedidos antes destes serem reencaminhados (i.e. a tradução do pacote acontece imediatamente depois do mesmo chegar ao sistema).
  - OUTPUT - as regras desta cadeia são utilizadas para pacotes gerados pela própria *firewall*.

3.3.5 Sistema de Deteção de Intrusões

Um Sistema de Deteção de Intrusões (IDS) é constituído por um ou vários componentes que têm como função detetar comportamentos anómalos no sistema. Um IDS pode ser responsável por *controlar* uma rede, sendo um NIDS, ou um componente em específico, sendo um *Host Intrusion Detection System* (HIDS). Em

[2] http://www.frozentux.net/iptables-tutorial/iptables-tutorial.html

ambos os casos, o IDS deverá identificar quando estão a acontecer eventos anormais dentro do que se poderá considerar normal. Mediante esses eventos, o próprio componente poderá reagir (reativo), ou simplesmente informar o administrador de sistema (passivo).

Toda a ação e reconhecimento realizado por um IDS tem como base informação providenciada de antemão e, por essa razão, é necessário alguma fonte informativa que identifique um comportamento anormal. Existem dois tipos de IDS mediante a fonte de informação utilizada:

- Baseado em conhecimento - o IDS é suportado por uma base-de-dados com padrões de ataques e comportamentos anómalos bem conhecidos.
- Baseado em comportamento - o IDS aprende e é treinado para saber o que é o comportamento correto do sistema.

O snort [27] é um IDS para sistemas UNIX ou Windows e é o monitor da rede da Interface Díodo, com configuração para atuar como um NIDS. No contexto da Interface Díodo, o seu objetivo é verificar se existe a passagem (ou tentativa de passagem) de informação no sentido contrário ao correto funcionamento do díodo. Caso exista, este componente reage, reiniciando os gestores de projeção. O monitor é também ele virtualizado, ou seja, o NIDS é uma máquina virtual. Esta aproximação permite inseri-lo no ambiente virtual e, desta forma, monitorizar as redes de comunicação virtuais do ambiente virtual. O snort é baseado em conhecimento de regras que lhe permitem identificar padrões nos eventos anormais que acontecem no ambiente.

Em termos de deteção de intrusões, é ideal perceber quando alguma das réplicas está com um comportamento anómalo em relação às outras. Por exemplo, se existir um gestor de projeção com uma taxa de geração de tráfego superior a todas as outras, poderemos estar perante uma tentativa de Negação de Serviço (DoS). Este ataque tem como objetivo afetar a disponibilidade do sistema. O NIDS é um componente privilegiado para detetar este tipo de comportamento.

Além do clássico sistema de deteção de intrusões, têm-se realizado trabalhos para a realização de um detetor de intrusões virtual. Um NIDS quando virtualizado não tem uma visão completa do sistema. Além de não conseguir especificar qual máquina está a ter um comportamento anómalo, ele também não consegue atuar sobre as outras réplicas. O VNIDS [28], arquitetura para

detetar intrusões em ambientes virtuais, tem como objetivo colmatar o problema da fraca visão do sistema. O VNIDS é uma solução composta por um detetor de dados responsável por capturar dados da rede a partir das interfaces virtuais que compõem a comunicação ambiente virtual.

3.4 Sumário

A arquitetura da Interface Díodo divide-se em interfaces de entrada e saída, réplicas que gerem a projeção dos pacotes e um monitor que verifica o funcionamento do serviço. As tecnologias utilizadas pela Interface Díodo permitem simplificar a sua concretização uma vez que não é necessário reinventar a roda. O iptables como *firewall*, JPCap para captura de dados e o snort para deteção de comportamentos anómalos, permitem a construção de uma solução fiável e funcional.

Capítulo 4

Concretização

Neste capítulo é descrita a concretização da Interface Díodo, bem como uma descrição do fluxo de dados. Segundo a arquitetura definida anteriormente e as tecnologias utilizadas, as interfaces de entrada e saída são concretizadas recorrendo a configurações de *firewall*. Os gestores de projeção são os componentes mais complexos e sobre os quais se desenvolveu o maior trabalho de programação. O monitor, sendo concretizado utilizando snort, consiste em configurações de regras para a deteção de comportamentos anómalos.

4.1 Configurações de Rede

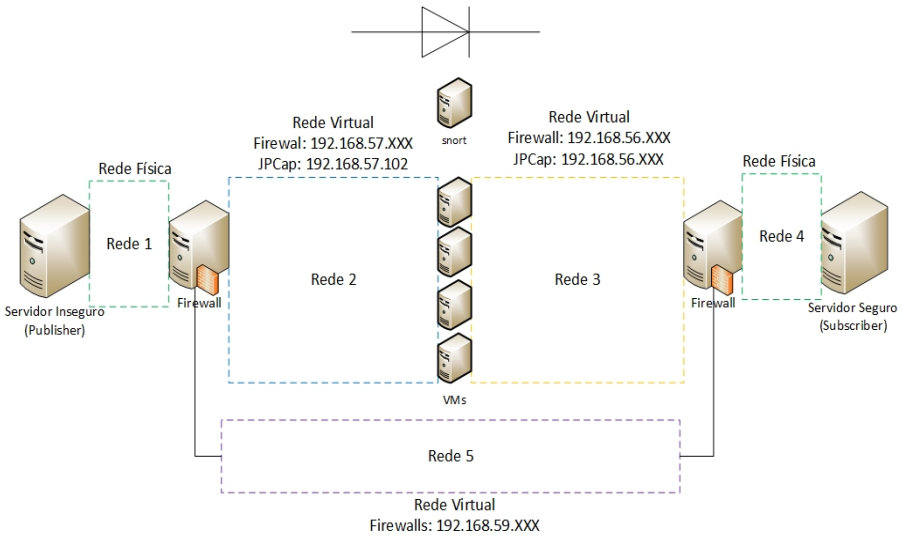

Figura 4.1: Redes virtuais.

A figura 4.1 apresenta o esquema de redes utilizadas e a tabela 4.1 as respetivas configurações. As entidades finais têm um endereço IP estático que é colocado no ficheiro de projeção da Interface Díodo. Desta forma, os pacotes que entrem em determinado porto da interface de entrada têm um endereço IP final ao qual se destinam. A *firewall* de entrada comunica em três redes: *Rede 1*, *Rede 2* e *Rede 3*. Os gestores de projeção comunicam em duas redes virtuais distintas, a *Rede 2* e *Rede 3*. Na *Rede 2*, os endereços IP das réplicas é o mesmo, para que quando a *firewall* de entrada reencaminhe um pacote, este seja entregue a todas as réplicas. A *firewall* de saída comunica através de três redes: *Rede 3*, *Rede 4* e *Rede 5*.

Componente	1	2	3	4	5
Externa	XXX.XXX.XXX.XXX	—	—	—	—
Firewall	YYY.YYY.YYY.YYY	192.168.57.101	—	—	192.168.58.101
Réplica 1	—	192.168.57.102	192.168.56.101	—	—
Réplica 2	—	192.168.57.102	192.168.56.102	—	—
Réplica 3	—	192.168.57.102	192.168.56.103	—	—
Réplica 4	—	192.168.57.102	192.168.56.104	—	—
Firewall Saída	—	—	192.168.56.105	YYY.YYY.YYY.YYY	192.168.58.102
Final	—	—	—	XXX.XXX.XXX.XXX	—

Tabela 4.1: Configurações de rede.

4.2 DiodePacket

A Interface Díodo concretiza um tipo de dados denominado por *DiodePacket*. Este pacote especial tem como objetivo substituir por completo o pacote original enviado por um emissor e criar um novo pacote específico à interface. Do pacote original retiram-se apenas os dados. Ao criar um novo tipo de dados, aproveitando apenas o campo de dados do pacote original, evitam-se ataques nos servidores finais, causados má formatação dos cabeçalhos dos pacotes.

Um pacote *DiodePacket*, cujos campos encontram-se na figura 4.2, desdobra-se em duas possibilidades: *DiodeTCPPacket* e *DiodeUDPPacket*. O *DiodeTCPPacket* destina-se a pacotes transportados pelo protocolo *Transmission Control Proto-*

col (TCP) e o DiodeUDPPacket a pacotes transportes por *User Datagram Protocol*(UDP). Estes novos objetos são composto pelos seguintes campos:

Campo	Descrição
ID	Identificador do pacote, colocado por cada réplica.
Flag	Para pacotes TCP, este campo indica se o pacote é de FIN
IP	Endereço IP do servidor final, retirado da tabela de projeção
Porto	Porto do servidor final que espera por pacotes de determinado serviço
Dados	Dados enviados pelo utilizador que serão entregues à aplicação final

Tabela 4.2: Campos do *DiodePacket*

4.3 *Firewalls*

A *firewall* de entrada e saída executam uma *firewall* iptables. Esta *firewall* tem dois objetivos: reencaminhar pacotes e bloquear a sua passagem quando enviados no sentido oposto da ligação.

A tabela 4.3 representa o ficheiro de configuração para a projeção de portos de entrada e dispositivos finais. Esta projeção indica à Interface Díodo o que fazer quando chega determinado pacote e para onde deve ser reencaminhado. No caso da *firewall* de entrada, esta tabela permite gerar de forma dinâmica as regras iptables segundo o protocolo de transporte utilizado bem como abrir os portos para receber informação.

Para cada entidade final (ou porto aberto), existe um conjunto de regras iptables, dependendo do protocolo de transporte, TCP ou UDP. Por exemplo, um servidor de *logs* tem o porto 514 à espera de dados transportados por UDP. A informação referente a esse servidor encontra-se num ficheiro de configuração que será lido para a criação de regras iptables.

Porto de Entrada	Protocolo	IP Final	Porto Final
514	udp	10.10.5.201	514
25	tcp	10.10.5.202	25
5555	tcp	10.10.5.203	5555
...	...	...	...

Tabela 4.3: Projeção de portos entre entrada da Interface Díodo e o dispositivo final.

4.3.1 *Firewall* de entrada

A *firewall* de entrada para ligações TCP e UDP (por esta ordem) é configurada com as seguintes regras:

```
iptables -t nat -A PREROUTING -p TCP --dport
    porto_servico -j DNAT --to-destination 192.168.57.
    YYY

iptables -t nat -A POSTROUTING -p TCP -d 192.168.57.
    YYY --dport porto_servico -j MASQUERADE

iptables -A INPUT -p TCP --tcp-flags PSH PSH -s
    192.168.57.YYY -j DROP

iptables -A OUTPUT -p TCP --tcp-flags RST RST -s
    192.168.57.YYY -j DROP
```

```
iptables -t nat -A PREROUTING -p UDP --dport
    porto_servico -j DNAT --to-destination 192.168.57.
    YYY
```

```
iptables -t raw -A PREROUTING -p UDP --sport
   porto_servico -j DROP
```

As duas primeiras regras, referentes a ligações TCP, recorrem à tabela de NAT. A primeira faz o reencaminhamento dos pacotes que chegam à Interface Díodo para os gestores de projeção. A segunda regra faz o reencaminhamento das respostas dos gestores de projeção para a entidade emissora. Esta segunda regra é necessária devido à bidirecionalidade do protocolo TCP, mais concretamente as respostas *ACK*. A terceira regra é responsável por rejeitar todos os pacotes de dados enviados pelos gestores de projeção, garantindo a unidirecionalidade do sistema. A quarta regra impede que a própria *firewall* cancele a ligação TCP com as réplicas centrais. Esta regra é necessária porque todos os gestores de projeção têm o mesmo IP. Esta necessidade deve-se ao fato de não ser possível realizar *broadcast* em ligações TCP. Atribuindo o mesmo endereço IP a todas as réplicas centrais é possível enviar um pacote para todas elas em simultâneo. Quando isso acontece, estas réplicas respondem com um *ACK*, que vão gerar um *RST*. É necessário bloquear este *RST* para que a ligação continue e, eventualmente, o emissor dos dados irá fechar a ligação.

As regras para ligações UDP são menos complexas devido à unidirecionalidade do protocolo. Assim sendo, é necessário definir uma regra que faça o reencaminhamento dos pacotes que chegam à *firewall* de entrada (primeira regra) e uma regra que bloqueie os dados enviados no sentido incorreto da ligação (enviadas pelos gestores de projeção, segunda regra).

4.3.2 *Firewall* de saída

A realização da *firewall* de saída é constituída por duas partes: componente iptables e concretização em Java. Relativamente à componente iptables, e visto que a *firewall* de saída não faz um reencaminhamento imediato dos pacotes, apenas existem regras de bloqueamento. Estas regras são também elas geradas segundo o ficheiro de projeção, uma vez que este contém informação sobre os dispositivos finais. Assim sendo, para cada entidade final, a *firewall* de saída tem as seguintes regras iptables:

```
iptables -A INPUT -p TCP --tcp-flags PSH PSH -s
    ip_maquina_final -j DROP
```

```
iptables -t raw -A PREROUTING -p UDP --sport
    porto_maquina_final -j DROP
```

A primeira regra bloqueia todo o tráfego gerado pelas entidades finais através de TCP e, a segunda, bloqueia todo o tráfego enviado através de UDP.

O componente Java e o fluxo de dados desde a chegada de um pacote à *firewall* de saída e o envio para a entidade final está representado na figura 4.2. Cada pacote enviado por um gestor de projeção é colocado numa fila, na posição correspondente ao identificador que traz. Para cada identificador são guardados os pacotes das diferentes réplicas. A fila é concretizada através de um *HashTable* e mantém os pacotes até chegar à sua vez de envio e o número de pacotes para esse identificador ser igual ou superior $2f + 1$. Quando existem $2f + 1$ pacotes iguais significa que existe uma maioria. Desta forma é possível à *firewall* de saída identificar um pacote correto e enviá-lo para o dispositivo final.

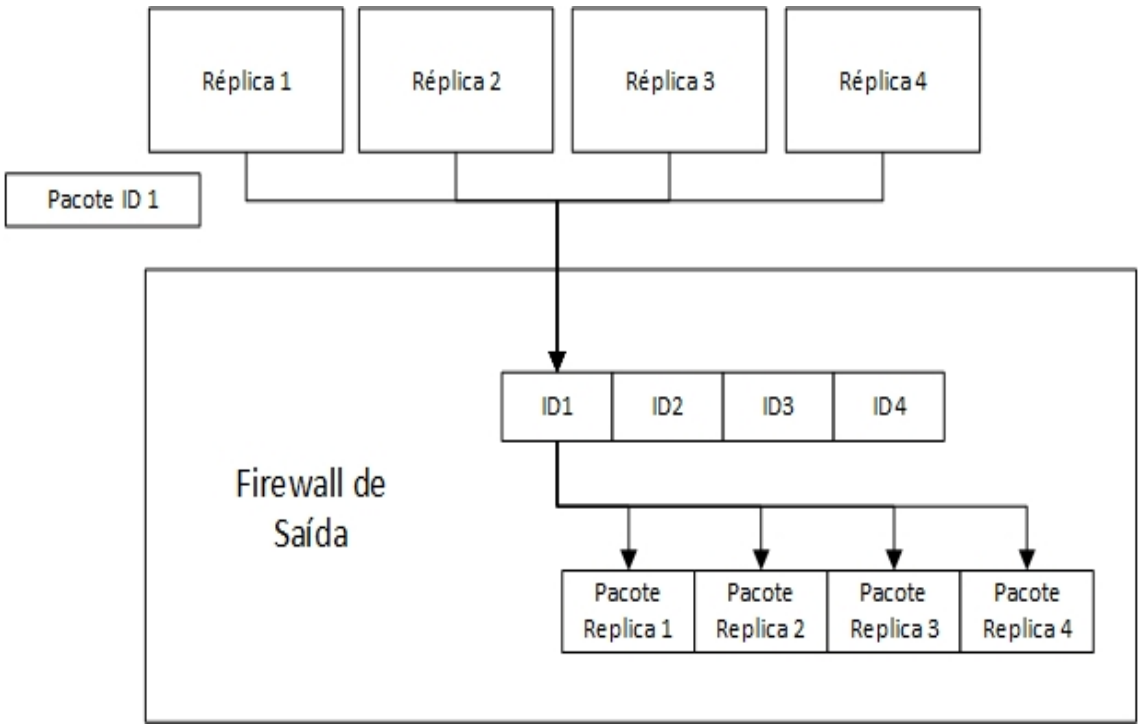

Figura 4.2: Fila da *firewall* de saída

Quando existem na fila $2f + 1$ pacotes iguais para o atual ID, a *firewall* de saída envia esse pacote para o dispositivo final. A ligação com o dispositivo final depende do protocolo de transporte que fez chegar o pacote até aqui. Se o pacote chegou à Interface Díodo através de TCP, então será enviado pelo mesmo protocolo; se chegou com UDP, será utilizado UDP. Assim sendo, a *firewall* de saída tem um repositório de *sockets* para ligações TCP ativas e cria uma nova ligação para cada pacote transportado por UDP. Quando chega à *firewall* de saída um pacote transportado por TCP e destinado a um dispositivo cuja informação ainda não se encontra no repositório de *sockets*, é criada uma ligação com o dispositivo final que é guardada. Desta forma, sempre que chegar um pacote destino a esse dispositivo facilmente se reaproveita a ligação estabelecida anteriormente.

Sempre que um gestor de projeção é reiniciado durante o tempo de operação da Interface Díodo é necessário passar-lhe o estado atual do sistema. Este estado é composto pelo ID do próximo pacote esperado. É responsabilidade da *firewall* de saída passar o estado à réplica uma vez que está numa posição privilegiada para obter essa informação. Sempre que é necessário passar o estado atual dos pacotes para uma réplica, o sistema abranda de forma a que as outras réplicas em funcionamento não avancem de forma rápida (incrementando rapidamente o ID atual). Esse abrandamento é realizado pelo canal escondido entre a *firewall* de entrada e saída. A *firewall* de saída envia um sinal bem definido à *firewall* de entrada e esta abranda a velocidade de reencaminhamento de pacotes. Quando a réplica obtém o ID atual, a *firewall* de saída envio outro sinal à *firewall* de entrada para que esta restabeleça o normal funcionamento do sistema. Este controlo de fluxo também é realizado quando a fila ultrapassa um limite pré-definido, situação que acontece quando o dispositivo final é mais lento que a Interface Díodo.

4.4 Gestor de Projeção

Os gestores de projeção têm duas funções: capturar os pacotes reencaminhados pela *firewall* de entrada utilizando a biblioteca JPCap e criar novos pacotes próprios da Interface Díodo. Podemos assumir que estas réplicas *cortam* o fluxo de dados provenientes das componentes de trás (rede externa e *firewall*) e criam um novo fluxo de dados com pacotes próprios da interface.

4.4.1 Fluxo de Dados

Sempre que o JPCap captura um novo pacote, é retirado o campo de dados e criado um pacote *DiodePacket*. Dependendo se o protocolo de transporte é TCP ou UDP, cria-se um *DiodeTCPPacket* ou *DiodeUDPPacket*. O ID colocado pelas réplicas depende do seu ID atual. Este ID atual corresponde ao número de pacotes recebido, ou seja, por cada novo pacote o valor é incrementado. O endereço IP e porto da entidade final ao qual estes pacotes se destinam são verificados na tabela de projeção e são colocados no novo objeto.

De forma sucinta, e segundo a tabela 4.3, o que chega à Interface Díodo no porto 514 é para ser entregue à entidade final com IP 10.10.5.201 no porto 514. Quando determinado pacote chega aos gestores de projeção, estes consultam o porto do pacote original e procuram-no na tabela de projeção, na coluna *Porto de Entrada*. Estes portos terão que estar abertos na *firewall* de entrada, significando que pacotes para outros portos não chegam aos gestores de projeção. Encontrando o porto original tabela de projeção, as réplicas centrais retiram o IP e porto do dispositivo final. Posteriormente, é criado o objeto *DiodePacket* com esta informação e é enviado para a *firewall* final.

A comunicação entre os gestores de projeção e a *firewall* de saída é realizada por invocação de métodos remotos, mais precisamente através de Java RMI.

4.4.2 Recuperação Reativa

Os gestores de projeção são Bizantinos devido à sua complexidade. Quando o sistema de monitorização deteta que alguma destas réplicas está a fugir ao comportamento correto, dá-se início à recuperação. Entende-se por comportamento incorreto a tentativa de envio de dados no sentido inverso ao correto funcionamento do díodo e quando uma réplica tem uma taxa de transferência muito superior ou inferior das restantes réplicas. Quando uma réplica é reiniciada e volta à operação normal envia um pedido de estado à *firewall* final. A *firewall* de saída informa a de entrada para abrandar o envio de pacotes. Quando o sistema abranda, a réplica recuperada recebe o estado (ID) atual. Uma vez recebido o ID, o sistema normaliza e todas as réplicas partem do mesmo estado.

Estas réplicas são as únicas componentes do sistema que executam *software* suscetível a vulnerabilidades. Esta afirmação deve-se ao fato de não existir in-

formação que confirme ou negue a existência de vulnerabilidades no JPCap. Por essa razão, as réplicas JPCap consideram-se Bizantinas e podem assumir um comportamento arbitrário: podem enviar pacotes errados, omitir pacotes ou parar o seu funcionamento. Para colmatar esta possibilidade existem $3f + 1$ réplicas deste tipo, sendo f o número de faltas que o sistema tolera.

4.5 snort

O snort tem sobre sua alçada o controlo do sistema, mais concretamente o comportamento dos gestores de projeção. A sua função é detetar tráfego no sentido inverso da ligação e possíveis ataques de Negação de Serviço (DoS). Sempre que um gestor de projeção *tenta* enviar dados no sentido incorreto de funcionamento ou tem uma taxa de transferência substancialmente acima ou abaixo das restantes réplicas, o snort inicia o processo de reinicialização.

Sempre que algum comportamento corresponde a uns dos padrões definidos nas regras do snort, este gera um *log* de controlo. Isto significa que sempre que o *log* for alterado, acontece algo no sistema que se encaixa num (ou vários) comportamentos incorretos definidos pelas regras do snort. Assim sendo, através da verificação da modificação destes ficheiros de *log* é possível saber se aconteceu algo de errado. É precisamente este processo que a Interface Díodo realiza. Retiraram-se todas as regras pré-definidas do snort e acrescentámos um novo pacote de regras específicas para a Interface Díodo. As regras são:

```
alert tcp 192.168.57.102 any -> 192.168.57.101 any (
   flags: P; message: Tentativa de envio de dados TCP
   no sentido incorreto);

alert udp 192.168.57.102 any -> 192.168.57.101 any (
   message: Tentativa de envio de dados UDP no sentido
    incorreto);
```

A primeira regra alerta (e escreve no *log*) sobre tentativas de tráfego TCP entre as réplicas centrais e a *firewall* de entrada. Não é possível evitar qualquer tipo de interação TCP uma vez que quando o tráfego chega à Interface Díodo através deste protocolo são os gestores de projeção que respondem com os ACK, embora

para o exterior essa geração de confirmações seja criada pela *firewall* de entrada. Por essa razão, apenas as interações de *push* (envio de dados, *flags: P* na regra acima) são detetadas pelo *Network Intrusion Detection System* (NIDS). Como também podem existir tentativas de interação no sentido inverso do funcionamento da Interface Díodo utilizando UDP, existe a segunda regra para detetar esse comportamento. Quando algum comportamento no sistema se encaixa numa uma destas regras, o snort, através do *host OS*, reinicia o gestor de projeção.

Esta concretização não é perfeita. Só o seria se a ação de reiniciar fosse realmente executada pelo snort. A razão pela qual isto não é possível deve-se ao fato de o snort ser ele próprio uma máquina virtual que se encontra dentro do mesmo ambiente virtual das outras réplicas. Isto significa que o sistema de monitorização não tem conhecimento da identificação das outras réplicas nem privilégios para as reiniciar. Esta limitação impede, também, que o snort identifique uma tentativa de DoS quando uma réplica central maliciosa envia uma quantidade substancial de mensagens porque não tem sensibilidade de unidade. Isto significa que o snort não consegue saber especificamente que existe uma e só uma réplica a gerar mais tráfego do que as outras, nem de qual réplica se trata. Assim sendo, existe um *software* mínimo no *host OS* responsável por fornecer um método remoto que será invocado pelo snort. Quando o snort invoca o método para reiniciar, este *software* executa uma primitiva do *hypervisor* para reiniciar cada uma das réplicas centrais. Para cada réplica central, é executado o seguinte comando:

```
VBoxManage controlvm $nome_maquina reset
```

Quando uma máquina é reiniciada, ela volta ao estado operacional com um novo sistema operativo. Desta forma garante-se diversidade de sistemas operativos, uma boa prática em termos de segurança.

4.6 Algoritmo

Relativamente à *firewall* de entrada, foi desenvolvido um módulo para receber os pedidos de abrandamento provenientes da *firewall* de saída. A comunicação entre *firewalls* é realizada através do canal escondido e por invocação a métodos remotos (Java RMI). Quando o método é invocado pela *firewall* de saída, é exe-

cutado um comando shell no *Guest OS* que abranda ou normaliza a velocidade de transmissão de dados.

Em termos gerais, a concretização Java realizada encontra-se nos gestores de projeção, *firewall* de saída e *Host OS*. Nos gestores de projeção é utilizado a biblioteca Java do JPCap para capturar pacotes e, posteriormente, criam-se objetos específicos (*DiodePacket*, *DiodeTCPPacket* e *DiodeUDPPacket*) para dar continuidade ao fluxo de dados. Estes pacotes são enviados por cada uma das réplicas para a *firewall* de saída que os irá colocar temporariamente numa lista, na posição do ID atual, até que existam $2f + 1$ pacotes para esse ID. Quando atingido esse número, a *firewall* de saída envia os dados para a entidade final.

O Algoritmo 1 descreve a componente de concretização Java da *firewall* de saída. O algoritmo (e realização) da *firewall* de saída inside essencialmente no tratamento da fila de pacotes e o envio dessa informação para o dispositivo final. Das linhas 1 - 14 dá-se a inicialização das variáveis necessárias para a concretização da *firewall* de saída. Na linha 1 é criado o ID que faz o controlo dos pacotes já enviados. Na linha 2 é inicializada a estrutura que irá manter os pacotes provenientes dos gestores de projeção até serem enviados para a réplica final. Na linha 5 é inicializada a estrutura que manterá as ligações (*sockets*) (TCP) abertas e das linhas 7 - 12 são criadas as ligações.

A função $RECEIVE(pacote)$ (linhas 15 - 20) é responsável por fazer o tratamento do pacote quando este chega à *firewall* final. Quando chega um pacote, este é colocado numa *HashTable* mediante o seu ID. Se na inserção deste pacote a fila ultrapassar um tamanho pré-determinado, a *firewall* de saída *notifica* a *firewall* de entrada para abrandar o reencaminhamento de pacotes.

Nas linhas 21 - 24 é apresentada a remoção de pacotes. Esta remoção é realizada utilizando o ID do último pacote enviado para o servidor final. Sempre que um conjunto de pacotes com o mesmo ID é removido da lista, é necessário incrementar a variável que faz a contagem do número de pacotes removidos. Esta variável é importante porque quando uma máquina recupera, esta pede o ID do último pacote enviado pelas réplicas centrais. O ID do último pacote enviado pelas réplicas centrais é o tamanho atual da lista mais o número de pacotes já removidos.

A função $GetPacket()$ é responsável por *escolher* o pacote correto para determinado ID. Ou seja, para determinado ID existem, pelo menos, $2f + 1$ pacotes.

Desses pacotes, f podem ser maliciosos. Quando for selecionado um dos pacotes corretos, este é enviado para o dispositivo final (descrito na função $main()$).

Algoritmo 1 Firewall de Saída

```
1:  function INITIALIZE()
2:      id ← 1
3:      queue[pacote.id] ← ∅
4:      removidos ← 0
5:      tcpPool ← ∅
6:      ligacaoUdp
7:      for all servidorFinal do
8:          if servidorFinal.protocolo == tcp then
9:              tcpPool ← [servidorFinal.ip][serivodorFinal.port]
10:         else if servidorFinal.protocolo == udp then
11:             ligacaoUdp ← [servidorFinal.ip][serivodorFinal.port]
12:         end if
13:     end for
14: end function
15: function RECEIVE(pacote)
16:     queue[pacote.id] ← pacote
17:     if queue.size >= MAXSIZE then
18:         SLOWDOWN()
19:     end if
20: end function
```

Nas linhas 33 - 35 está a concretização da recuperação de estado. Para o efeito, a *firewall* de saída indica à de entrada para abrandar o envio de pacotes para que a passagem de estado seja efetuada com sucesso. Desta forma, pode dizer-se que o sistema pára e as informações necessárias que influenciam o estado a ser enviado não são modificadas.

```
21: function GetPacket()
22:     if queue[id].size() >= 2f + 1 then
23:         pacote ← queue[id]
24:         id ← id + 1
25:         return pacote
26:     end if
27:     return null
28: end function
29: function Remove()
30:     queue[id].remove()
31:     removidos ← removidos − 1
32: end function
33: function GetLastMessageId()
34:     FirewallEntrada.slowDown()()
35:     return removidos + queue.size()
36: end function
37: function SlowDown()
38:     FirewallEntrada.slowDown()()
39: end function
40: function main()
41:     while true do
42:         paraEnviar ← GetPacket()
43:         if paraEnviar ≠ null then
44:             if paraEnviar.protocolo == tcp then
45:                 socket ← tcpPool[toSend.ip]
46:                 socket.send(paraEnviar)
47:                 Remove()
48:             else if paraEnviar.protocolo == udp then
49:                 datagram.send(paraEnviar)
50:                 Remove()
51:             end if
52:         end if
53:     end while
54: end function
```

O Algoritmo 2 apresenta o pseudo-código da concretização Java dos gestores de projeção. Nas linhas 1 - 7 são inicializadas as variáveis utilizadas por cada instância. O *ID* representa o último *ID* colocado num pacote enviado para a *firewall* de saída. Se a réplica nunca falhou e recuperou, significa que em qualquer altura no tempo este *ID* é igual ao número de pacotes que a réplica recebeu. Se a réplica falhar, quando recupera este *ID* é obtido através de um pedido à *firewall* de saída.

A *tabelaRouting* é a estrutura que mantém a informação relativa à projeção entre servidores emissores e servidores finais. As réplicas consultam esta tabela para construir os objetos Java que serão enviados à *firewall* de saída.

O modo de operação dos gestores de projeção consiste em ter o módulo de captura do JPCap ativo. Este *JPCap.capture*() captura todos os pacotes que chegam à interface de rede definida e destinados aos portos pré-definidos. Estes portos são especificados no JPCap bem como os protocolos de transporte utilizados na transmissão dos pacotes. Assim sendo, para cada entrada na tabela de projeção são criados filtros no JPCap e, desta forma, restringe-se a captura à informação que se pretende dar seguimento. Quando é capturado um novo pacote, verifica-se se o protocolo de transporte utilizado é TCP (linha 11) ou UDP (linha 18). Se for TCP é criado uma nova estrutura de dados, *diodeTCPPacket* onde são colocadas as novas informações referentes ao pacote e os dados originais. Se se trata de UDP, é criado um *diodeUDPPacket* e o restante procedimento é idêntico. Quando é enviado um pacote para a *firewall* de saída, o *ID* é incrementado.

Algoritmo 2 Gestores de Projeção

```
 1: function INITIALIZE()
 2:     id ← GETLASTMESSAGEID()
 3:     tabelaRouting ← ∅
 4:     for all servidorFinal do
 5:         tabelaRouting ← [servidorFinal.ip][servidorFinal.port]
 6:     end for
 7: end function
 8: function MAIN()
 9:     while true do
10:         pacote ← JPCaptor.capture()
11:         if pacote.protocolo == tcp then
```

```
12:             diodeTCPPacket.id ← id
13:             diodeTCPPacket.port ← tabelaRouting[pacote.port]
14:             diodeTCPPacket.ip ← tabelaRouting[pacote.port]
15:             diodeTCPPacket.data ← tabelaRouting[pacote.data]
16:             SEND(diodeTCPPacket)
17:             id ← id + 1
18:         else if pacote.protocolo == udp then
19:             diodeUDPPacket.id ← id
20:             diodeUDPPacket.port ← tabelaRouting[pacote.port]
21:             diodeUDPPacket.ip ← tabelaRouting[pacote.port]
22:             diodeUDPPacket.data ← tabelaRouting[pacote.data]
23:             SEND(diodeUDPPacket)
24:             id ← id + 1
25:         end if
26:     end while
27: end function
```

4.7 Sumário

Este capítulo descreve os desafios associados à Interface Díodo bem como a realização e o fluxo da informação dentro do dispositivo. É apresentada uma configuração de rede que torna percetível o modo de funcionamento interno do ambiente virtual. A Interface Díodo é composta, essencialmente, por duas *firewalls*, uma de entrada e outra de saída, réplicas centrais responsáveis por receber o tráfego exterior e criar um novo tipo de dados Díodo, e uma réplica snort para detetar comportamentos anormais.

Para cada componente do sistema é descrito o seu funcionamento e a sua configuração para que, em conjunto, possam criar uma analogia com o díodo da Física. Esta solução é totalmente realizada em *software*, em contraposição com as demais soluções que se dizem concretizadas em *hardware* embora recorram, obrigatoriamente, a *software*.

Capítulo 5

Resultados

Este capítulo descreve a fiabilidade da concretização, o ambiente onde foram executados os testes do protótipo concretizado, bem como as variáveis tidas em conta e os resultados de desempenho obtidos.

O serviço fornecido pela Interface Díodo pode dividir-se em dois procedimentos base: transportar os dados entre emissor e serviço final e bloquear o tráfego no sentido oposto. O primeiro procedimento está diretamente ligado ao sucesso ou insucesso da prestação do serviço fornecido pela entidade final, por exemplo serviço de e-mail, *logs*, votação *online*, e todos os outros serviços unidirecionais. Se os dados não forem devidamente transmitidos, então o serviço falha. Perante este contexto, e depois de descrito o modo de funcionamento da interface, concluímos que apenas falhas na rede podem causar problemas no cumprimento deste objetivo. O modo de operação da Interface Díodo consiste em copiar os dados dos pacotes que chegam, colocá-los num novo pacote e enviar para a entidade final. Isto significa que todos os serviços unidirecionais funcionam, de fato, com a nossa concretização.

Como não foi possível testar a concretização recorrendo aos vários serviços passíveis de testes, escolhemos um deles para atestar o funcionamento da Interface Díodo. O caso de uso escolhido foi o serviço de *logging*. As razões para esta escolha estão relacionadas com a complexidade de configuração dos outros serviços. Por exemplo, para testar um serviço de e-mail seria necessário configurar um servidor de e-mail.

Para adicionar um novo serviço à Interface Díodo basta acrescentar uma entrada ao ficheiro de projeção. Essa entrada deverá conter o porto de entrada da

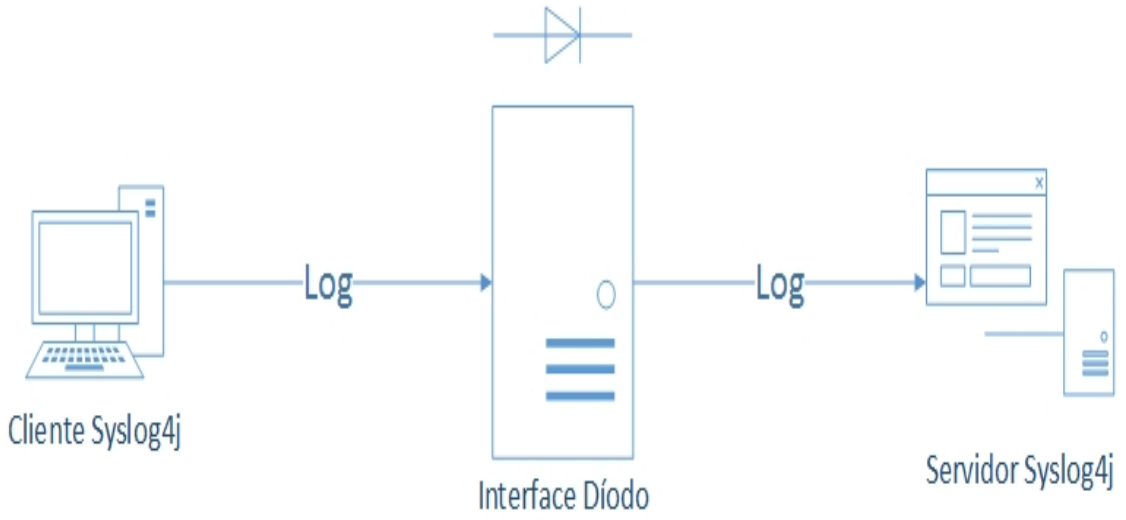

Figura 5.1: Interação com Syslog.

interface para o serviço em questão, o protocolo de transporte, o IP e o porto da entidade final, como representado na figura 4.3. Sempre que se adicionar uma nova entrada ao ficheiro de projeção é necessário reiniciar o díodo para que sejam geradas as regras iptables referentes à nova entrada e sejam atualizadas as estruturas de dados dos gestores de projeção.

Os testes foram executados utilizando uma biblioteca Java, Syslog4j, para geração de *logs* Syslog. Esta biblioteca permite executar clientes geradores de *logs* que são posteriormente enviados para um servidor. O servidor recebe os *logs* e mostra-os ao utilizador. A configuração do cliente Syslog4j consiste em fornecer o endereço IP e o porto do servidor final. Neste caso, o endereço IP e o porto do *servidor final* são os dados referentes à Interface Díodo. O servidor é configurado fornecendo o porto sobre o qual os *logs* são recebidos. A figura 5.1 representa a interação entre o Syslog4j e a Interface Díodo.

A figura 5.2 representa o processamento simples de quatro réplicas que geram a projeção de vinte *logs*, cada um deles com 1000 bytes. As primeiras linhas representam o arranque dos gestores de projeção, e como indicado na figura, o que chega ao porto 5555 será entregue à máquina final 10.10.5.221 no porto 5555 (caso os dados sejam transportados por TCP) e os dados que chegaram no porto 514 (por UDP) serão entregues à mesma máquina final no porto 7777. A partir desta altura, cada réplica cria um novo pacote de dados e envia-o para a Interface de Saída.

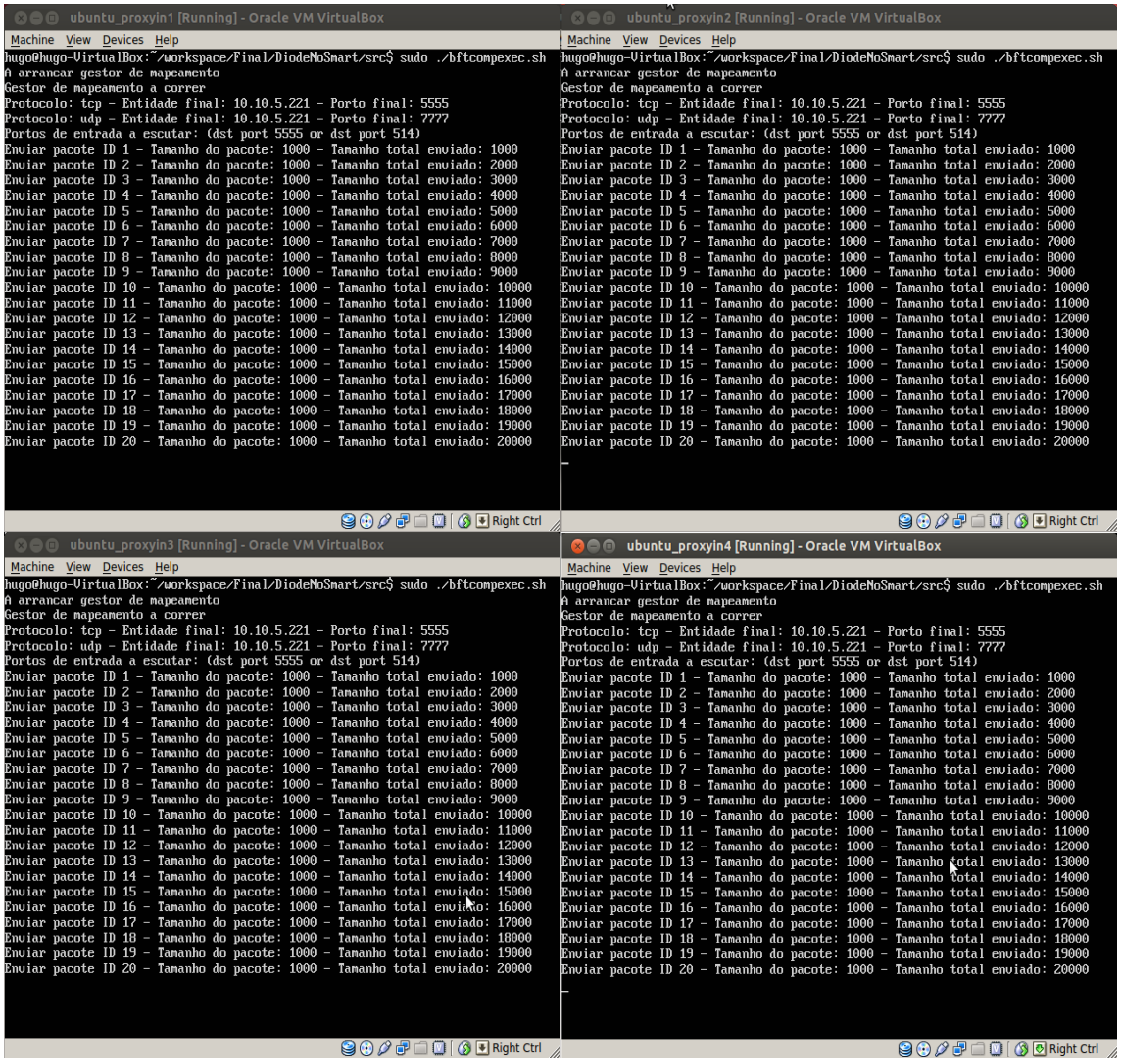

Figura 5.2: Processamento nos gestores de projeção.

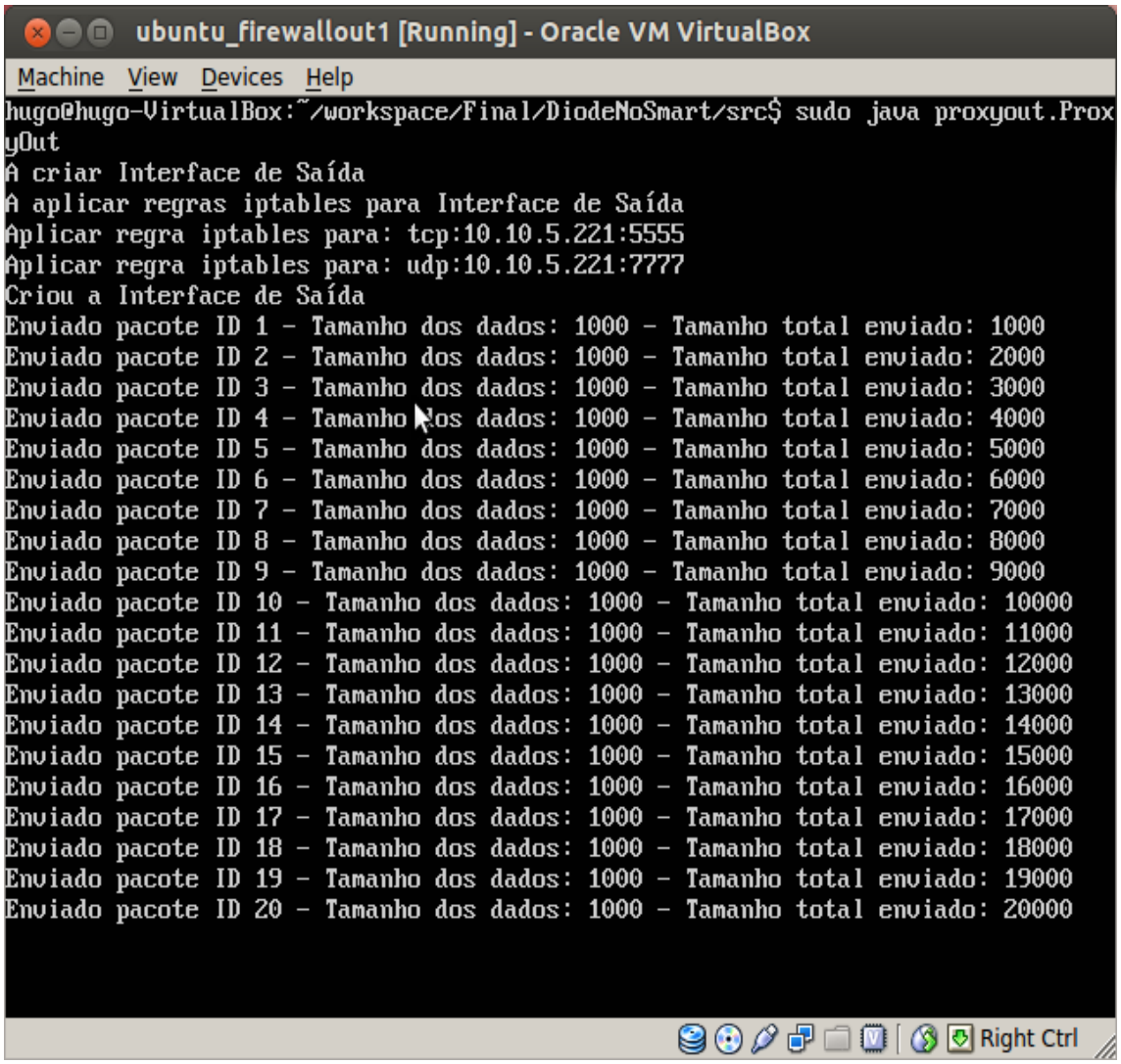

Figura 5.3: Processamento na Interface de Saída.

Na figura 5.3 está representado o envio dos dados para as entidades finais. A interface de saída recebe, pelo menos, $2f + 1$ pacotes de dados provenientes dos gestores de projeção e, nessa altura, tem a informação necessária para selecionar o pacote a enviar uma vez que tem uma maioria de pacotes corretos. Depois dessa seleção, o pacote é enviado para a entidade final, conforme demonstrado na figura 5.3.

Depois da concretização dos componentes descritos anteriormente, o resultado obtido foi uma Interface Díodo com um funcionamento correto. Assim se conclui que é possível concretizar um díodo de dados totalmente baseado em *software*. Os testes foram realizados num computador com processador Core 2

Duo, 4GB de RAM e o *software* de virtualização Oracle VM VirtualBox. Neste computador foram executadas sete máquinas virtuais que compõem o protótipo: uma *firewall* de entrada, quatro gestores de projeção, uma *firewall* de saída e, por fim, uma réplica a executar snort. As máquinas virtuais executam Ubuntu como *guest OS*.

A tabela 5.1 representa a entrada no ficheiro de projeção referente ao serviço de Syslog para a realização dos testes. Para entender o desempenho da Interface Díodo realizou-se uma bateria de testes. Estes testes consistem no envio de *logs* a partir de um cliente para um servidor Syslog4j, sem o díodo a mediar a transmissão. Depois realizaram-se os mesmos envios com a Interface Díodo a mediar a transmissão dos dados. Cada *log* é composto por 1000 bytes de dados.

Porto de Entrada	Protocolo	IP Final	Porto Final
7777	udp	10.10.5.221	7777

Tabela 5.1: Entrada no ficheiro de projeção referente ao serviço de Syslog4j.

	1 MB (cada cliente)		2 MB (cada cliente)		3 MB (cada cliente)	
	S/ Díodo	C/ Díodo	S/ Díodo	C/ Díodo	S/ Díodo	C/ Díodo
1 Cliente	3,1 seg.	5,1 seg. (+65%)	5,7 seg.	8,9 seg. (+56%)	8,2 seg.	13,4 seg. (+63%)
2 Clientes	5,2 seg.	7,2 seg. (+38%)	8,8 seg.	13,3 seg. (+51%)	13,5 seg.	20,1 seg. (+49%)
3 Clientes	6,9 seg.	9,9 seg. (+45%)	13,1 seg.	19,7 seg. (+50%)	19,8 seg.	30,2 seg. (+52%)

Tabela 5.2: Desempenho da Interface Díodo.

A tabela 5.2 e o gráfico 5.4 mostram o desempenho da Interface Díodo em comparação às mesmas operações sem recorrer à interface. Os tempos obtidos são referentes ao envio de 1MB (1000 *logs* de 1000 bytes) por cada cliente, 2MB (2000 logs de 1000 bytes cada) por cada cliente e 3MB (3000 *logs* de 1000 bytes cada) por cada cliente. Como é possível verificar, a comunicação mediada pela Interface Díodo, quando comparada com a comunicação sem a interface, piora o desempenho do envio de *logs* entre 40 - 65 por cento. Por exemplo, para o envio de mil pacotes de 1000 bytes por um cliente (1MB), a Interface Díodo demorou mais 65 por cento do tempo que o envio direto. A diferença no desempenho com

e sem díodo é relevante e deve-se à complexidade inserida no protocolo. Com especial atenção ao processo de captura de pacotes nos gestores de projeção e ao processo de escolha do pacote correto na fila da Interface de Saída. O processo de escolha do pacote correto envolve, para cada pacote, $n - 1$ comparações no pior caso, sendo n o número de gestores de projeção e f o número de réplicas Bizantinas. Para uma fila com k IDs, a complexidade do envio do pacote correto para a máquina final é $O(n * k)$. Isto significa que para enviar mil pacotes são realizadas quatro mil operações extra quando comparado com a transmissão sem a Interface Díodo.

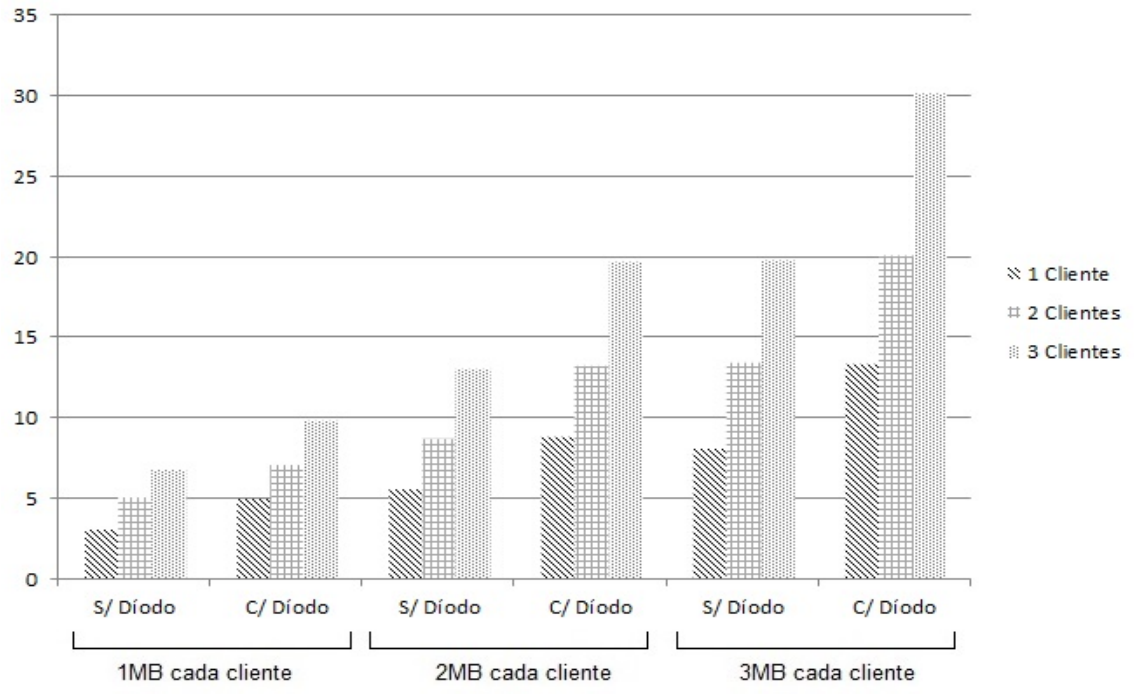

Figura 5.4: Gráfico de desempenho da Interface Díodo.

Além das operações extra, o processo de captura e manipulação de pacotes através de um biblioteca para o efeito é mais lento que a leitura direta nos *buffers* dos *sockets* que recebem os dados. No entanto, a manipulação em si é realizada com maior facilidade e, por outro lado, se for uma tecnologia bem conhecida, a possibilidade de introdução de vulnerabilidades também é reduzida.

A tabela 5.3 e o gráfico 5.5 apresentam o número de *logs* transmitidos por segundo. A discrepância do número de *logs* transmitidos por segundo entre a comunicação com e sem Interface Díodo deve-se ao fato de na primeira, mesmo

que a interface receba um determinado *log* (por exemplo, o *log* com $ID\ n$), este permanecerá na fila da interface de saída depois de enviados os $n-1$ *logs* que o antecedem. Ou seja, independentemente da taxa de entrada na interface (o cliente pode transmiti-los a uma velocidade considerável), o seu despacho está sempre dependente do envio dos *logs* anteriores. Quando a comunicação é realizada sem a interface não existe esta dependência.

	1 MB (cada cliente)		2 MB (cada cliente)		3 MB (cada cliente)	
	S/ Díodo	C/ Díodo	S/ Díodo	C/ Díodo	S/ Díodo	C/ Díodo
1 Cliente	323 log/seg	196 log/seg	351 log/seg	225 log/seg	366 log/seg	224 log/seg
2 Clientes	385 log/seg	278 log/seg	455 log/seg	301 log/seg	444 log/seg	299 log/seg
3 Clientes	435 log/seg	303 log/seg	458 log/seg	305 log/seg	455 log/seg	298 log/seg

Tabela 5.3: Número de *logs* por segundo.

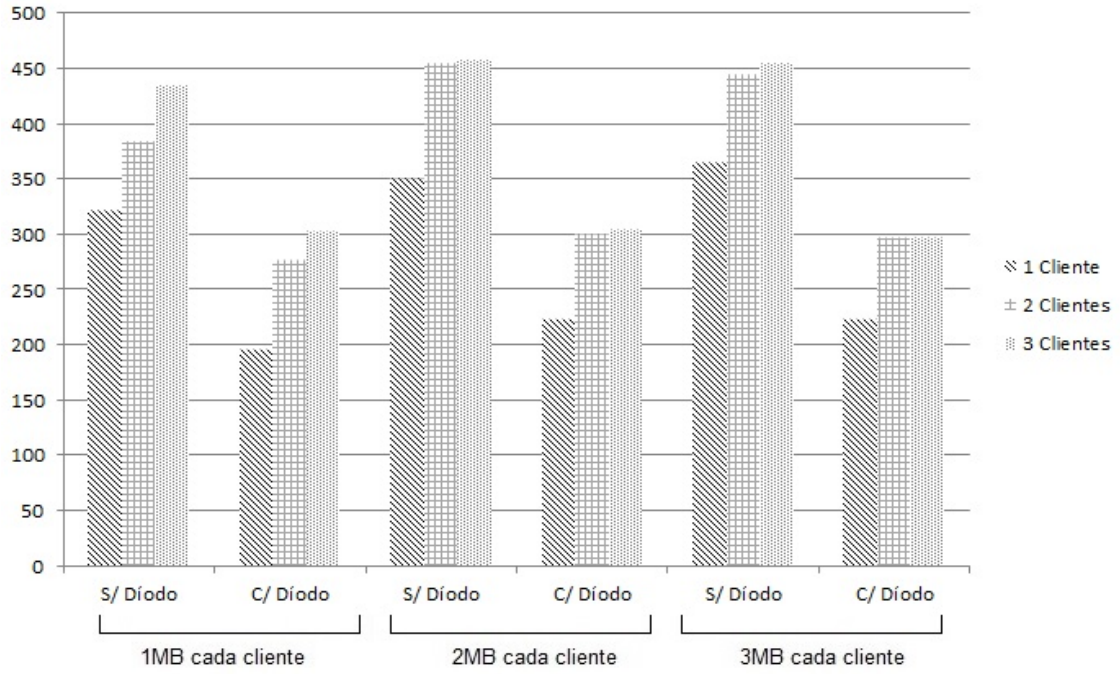

Figura 5.5: Gráfico do número de *logs* por segundo.

O *trade-off* entre a utilização de uma ferramenta que permita a fácil manipulação dos pacotes recebidos por um dispositivo ou a leitura direta dos *buffers* dos *sockets* é interessante e, depois de realizada a concretização e os testes, conclui-se

que será mais eficaz recorrer diretamente aos *buffers* no caso de comercialização da Interface Díodo. A atual concretização não recorre diretamente aos *buffers* dos *sockets* porque quando chegámos à conclusão que trariam um melhor desempenho já era demasiado tarde para realizar as alterações. No entanto, não sabendo como o JPCap opera no processo de captura de pacotes até apresentá-los ao utilizador em forma de objeto Java, com certeza que é um processo mais lento que a leitura direta dos pacotes no *socket*.

Outra medida para melhorar o desempenho da Interface Díodo é o desenvolvimento do mesmo serviço recorrendo a técnicas de *batching*. É um desafio interessante porque não é trivial concretizar técnicas de *batching* com ligações TCP. Aquando a utilização do protocolo UDP, a técnica de *batching* torna-se simples de realizar uma vez que não existe qualquer significado lógico de ordem em termos de transporte entre dois pacotes consecutivos.

5.1 Desafios e Limitações da Concretização

O processo de construção do protótipo da Interface Díodo passou por diversas fases e experimentações. Grande parte do trabalho incidiu sobre as réplicas centrais e a sua lógica de funcionamento. A ideia nunca foi reinventar a roda. Existe um conjunto de tecnologias cujos serviços e mecanismos que fornecem, em conjunto, permitem criar a analogia do díodo. Um dos trabalhos exploradores e experimentados foi o BFT-SMaRt [29], uma biblioteca de replicação de máquinas de estado tolerante a faltas Bizantinas. O BFT-SMaRt foi um excelente ponto de partida para o resultado final da Interface Díodo. Inicialmente fez sentido explorar o BFT-SMaRt e usá-lo no protótipo. Depois de alguns testes realizados, chegou-se à conclusão que necessitávamos apenas de um algoritmo de *consensus*. Adaptámos a nossa concretização para utilizar somente a componente de *consensus* do BFT-SMaRt. Devido ao grande número de mensagens trocadas entre as réplicas através do algoritmo de *consensus* do BFT-SMaRt (para quatro réplicas centrais, eram trocadas dezasseis mensagens entre as réplicas), percebemos que era suficiente realizar uma consolidação no destino (*firewall* de saída), ou seja, cada uma das réplicas enviava os seus dados e a *firewall* de saída é responsável por *escolher* um dos dados corretos.

Neste trabalho abordou-se desde sempre as réplicas centrais como sendo

várias instâncias replicadas que executam o mesmo *software*. Isto significa que temos *n* réplicas a realizar o mesmo trabalho. Por outro lado, assumimos que as entidades extremas, ambas as *firewalls*, são seguras. Este pressuposto é plausível uma vez que são entidades simples, pouco complexas e bem conhecidas. Não obstante, também elas podem ser replicadas. Este trabalho explorou também essa possibilidade. No entanto, usar um esquema de replicação tanto para a *firewall* de entrada como de saída exige alguns cuidados e traz novos desafios.

O primeiro é o fato de a Interface Díodo lidar com o protocolo de transporte TCP. O TCP é um protocolo de transporte guiado à ligação, onde existe um fluxo de controlo entre emissor e transmissor. Se introduzirmos replicação nos extremos, é necessário garantir que um emissor, pelo período necessário de interação, recorre somente a uma das réplicas.

O segundo está relacionado com as respostas do gestores de projeção a confirmar a receção dos dados. Estas confirmações teriam que ser devolvidas pela *firewall* de entrada que fez o reencaminhamento. Isto significa que o melhor caminho a seguir será utilizar replicação para obter redundância, ou seja, caso uma réplica falhe existe outra para a substituir. Utilizar a replicação num esquema de balanceamento de carga pode trazer dificuldades acrescidas, embora seja um ótimo desafio.

Capítulo 6

Conclusão

Esta dissertação visa descrever uma implementação em *software* de um díodo de dados, um dispositivo de rede que permite a passagem de tráfego num só sentido. A Interface Díodo é composta por múltiplos componentes, responsáveis por, em conjunto, implementarem o protocolo unidirecional.

A contribuição deste trabalho envolve uma implementação alternativa e inovadora do díodo de dados, uma vez que as outras implementação são baseadas em *hardware*. As contribuições descritas neste trabalho são:

(a) **Implementação em *software*:** a Interface Díodo é implementada única e exclusivamente através de *software*;

(b) **Independência do protocolo de transporte:** a interface é independente do protocolo de transporte. Correto funcionamento tanto com TCP como com UDP;

(c) **Comunicação unidirecional:** existe um cliente (*publisher*) que produz informação que é escrita num servidor final (*subscriber*). Um produtor não pode ser simultaneamente um consumidor e vice-versa;

(d) **Tolerante a faltas Bizantinas:** as componentes replicados da Interface Díodo podem desviar-se do seu funcionamento correto ou serem atacadas. Porém, o sistema como um todo continua com um funcionamento correto;

(e) **Confidencialidade e Integridade**: os dados armazenados na entidade de alto nível de segurança são confidenciais, no caso do díodo diretamente polarizado, e íntegros no caso do díodo inversamente polarizado.

6.1 Trabalho Futuro

A duração do projeto desta dissertação foi suficiente para concretizar um protótipo simples e funcional. Devido às experiências realizadas ao longo deste período para a obtenção da melhor solução, o tempo não permitiu desenvolver mais funcionalidades para a Interface Díodo. Atualmente, a Interface Díodo funciona com todas as aplicações que tenham um servidor final a correr o serviço correspondente à informação que está a ser trocada. Por exemplo, para que o envio de um *log* seja bem sucedido, deverá existir um dispositivo final a correr um serviço de *logging*. O próximo passo, e um caminho inteligente como trabalho futuro desta dissertação, é a implementação de aplicações (ou *plugins*) específicos da Interface Díodo. Assim sendo, poderá ser desenvolvido uma aplicação de e-mail, FTP, *logging* ou qualquer outra aplicação cuja operação faça sentido em termos de unidirecionalidade.

Outro desafio interessante, e que surgiu durante o período de realização desta dissertação, é a contextualização e materialização do conceito Interface Díodo para serviços de nuvem. Ou seja, fornecer um serviço Interface Díodo na nuvem (uma aplicação como serviço). Seria interessante idealizar e pensar sobre este tema, seguindo-se um estudo aprofundado sobre a viabilidade da sua implementação. Imagine o que poderia ser este conceito implementado na nuvem, onde alguns dos dispositivos que a compõem interagem através de uma aplicação Interface Díodo como serviço.

Um dos grandes desafios nos atuais mecanismos de segurança passa pela interpretação dos dados em trânsito. Essa interpretação, útil em *proxies*, é tão ou mais importante para um sistema como a Interface Díodo. Assim sendo, explorar a possibilidade dos gestores de mapeamento interpretarem os dados em trânsito seria uma mais-valia. Desta forma, a Interface Díodo não só funcionariam como um díodo, com tráfego unidirecional, mas também como analisador de tráfego, sendo possível detetar e bloquear dados que não fazem sentido para determinada aplicação.

Um dos aspetos mencionados neste trabalho foi a clara diferente entre um detetor de intrusões clássico e virtual. Visto que a Interface Díodo é maioritariamente virtualizada, uma vez que as suas réplicas são máquinas virtuais a executar dentro de um ambiente virtual, faz sentido colocar um detetor de intrusões também ele virtual. Porém, os trabalhos que apontam nesse sentido são

teóricos e existe pouco trabalho realizado num sentido prático.

Abreviaturas

ANS Alto Nível de Segurança

API Application Programming Interface

API Backreflection

BNS Baixo Nível de Segurança

DoS Denial of Service

FTP File Transfer Protocol

Guest OS Guest Operating System

HIDS Host Intrustion Detection System

HTTP Hypertext Transfer Protocol

IDS Intrustion Detection System

IP Internet Protocol

LAN Local Area Network

LLC Logical Link Control

MAC Media Access Control

NAT Network Address Translation

OSI Open System Interconnection

RPC Remove Procedure Call

SMTP Simple Mail Transfer Protocol

SSH Secure Shell

TCP Transmission Control Protocol

THP Trusted High Process

TLP Trusted Low Process

UDP User Datagram Protocol

VMM Virtual Machine Monitor

Bibliografia

[1] O. S. Zhiyong, L. Kui, Y. Shiping, and Qingbo, "Descriptive models for Internet of Things," in *Intelligent Control and Information Processing (ICICIP), 2010 International Conference on*, pp. 483 – 486, 2010.

[2] Y.-K. Chen, "Challenges and opportunities of internet of things," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pp. 383 – 388, 2012.

[3] Y. Zhang and J. Jiang, "Bibliographical review on reconfigurable fault-tolerant control systems," *Annual Reviews in Control*, vol. 32, pp. 229–252, Dec. 2008.

[4] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, July 1982.

[5] M. W. Stevens, "An Implementation of an Optical Data Diode," tech. rep., DSTO Electronics and Surveillance Research Laboratory, Salisbury,, 1999.

[6] M. Kang, I. Moskowitz, and S. Chincheck, "The Pump: A Decade of Covert Fun," *21st Annual Computer Security Applications Conference (ACSAC'05)*, pp. 352–360, 2005.

[7] M. Kang, I. Moskowitz, and D. Lee, "A Network Pump," *IEEE Transactions on Software Engineering*, vol. 22, pp. 329–338, May 1996.

[8] I. O. f. S. I. Electrotechnical, *ISO/IEC 7498-1*. 1994.

[9] Information Sciences Institute;, *Transmission Control Protocol/RFC: 793*. 1981.

[10] I. S. Institute, *User Datagram Protocol/RFC: 793*. 1980.

[11] B. Ramos, "Challenging Malicious Inputs with Fault Tolerance Techniques," pp. 1–8, 2007.

[12] D. Contractor, H. Side, L. Side, and D. Diode, "Secure, Unidirectional Data Flow with Network Taps," no. July, pp. 1–6, 2011.

[13] I. Link and P. Suite, "Interactive Link Data Diode System," tech. rep., 2013.

[14] H. Okhravi and F. Sheldon, "Data diodes in support of trustworthy cyber infrastructure," *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research - CSIIRW '10*, p. 1, 2010.

[15] M. Garcia, A. Bessani, and N. Neves, "Diverse OS Rejuvenation for Intrusion Tolerance," *IEEE/IFIP International Conference on Dependable Systems and Networks, Hong Kong, June 2011*, pp. 131–134, 2011.

[16] M. Garcia, N. Neves, and A. Bessani, "DIVERSYS : DIVErse Rejuvenation SYStem," *INFORUM - Simpósio de Informática (INFORUM), Lisboa, Portugal, September, 2012*, 2012.

[17] F. B. Schneider, "Replication Management using the State Machine Approach," vol. 4, 1990.

[18] S. N. T.-c. Chiueh, "A Survey on Virtualization Technologies," Tech. Rep. Vm, 2005.

[19] B. Pfaff and M. Rosenblum, "Terra : A Virtual Machine-Based Platform for Trusted Computing," *SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 193–206, 2003.

[20] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor : Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization," *SOSP '11 Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 203–216, 2011.

[21] P. M. Chen and B. D. Noble, "When Virtual Is Better Than Real," *2001 Workshop on Hot Topics in Operating Systems (HotOS)*, 2001.

[22] Z. Wang and X. Jiang, "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity," *2010 IEEE Symposium on Security and Privacy*, pp. 380–395, 2010.

[23] E. Keller, J. Szefer, and R. B. Lee, "NoHype : Virtualized Cloud Infrastructure without the Virtualization," *International Symposium on Computer Architecture (ISCA)*, pp. 350 – 357, 2010.

[24] P. Verissimo and L. Rodrigues, *Distributed Systems for System Architects*. Springer, 2001.

[25] Netfilter, "IPTABLES."

[26] Z. Robert and S. Suehring, *iptables : The Linux Firewall Administration Program*. Novell Press, 2006.

[27] Sourcefire, "Snort: A free lightweight network intrusion detection system for UNIX and Windows.," 2010.

[28] F. Zhao, W. Yang, H. Jin, and S. Wu, "VNIDS : A Virtual Machine-based Network Intrusion Detection System," *International Conference on Anti-counterfeiting, Security, and Identification (2008ASID)*, no. 2008, pp. 254 – 259, 2008.

[29] J. a. Sousa and A. Bessani, "From Byzantine Consensus to BFT State Machine Replication : A Latency-Optimal Transformation," *EDCC '12 Proceedings of the 2012 Ninth European Dependable Computing Conference*, pp. 37 – 48, 2012.

[30] B. W. Lampson, "A note on the confinement problem," *Communications of the ACM*, vol. 16, pp. 613–615, Oct. 1973.